



Welcome to FreeBASIC | Getting Help with FreeBASIC

Language Documentation

Keywords - Alphabetical
Keywords - Functional
Graphics Keywords
Operators List

Variables and Data Types

Variable Declarations
User Defined Types
Standard Data Types
Standard Data Type Limits
Converting Between Data Types

Operators

Operators
Operator Precedence
Bitwise Operators & Truth Tables

Statements

Control Flow
Procedures
Modularizing

Other

Preprocessor
Escape Sequences In String
Literals
Meta-statements
Intrinsic Defines
Error Handling

Tutorials

Programmer's Guide
Community Tutorials
Community Code Library
External Libraries Index

Using the FreeBASIC compiler

Installing FreeBASIC |
Requirements
Running FreeBASIC
Using the Command Line |
Command Line Options

Debugging with FreeBASIC

Compiler Error Messages
Tools used by fbc

FreeBASIC dialects and QBASIC

FreeBASIC and Qbasic |
Differences from QB
FreeBASIC Dialects

FAQs

Compiler FAQ
Graphics Library FAQ

Inline Asm

Runtime Library Reference

Array Functions
Bit Manipulation
Console Functions
Date and Time Functions
Error Handling Functions
File IO Functions
Mathematical Functions
Memory Functions
Operating System Functions
String Functions
Threading Support Functions
User Input Functions

Graphics Library Reference

2D Drawing Functions
User Input Functions
Screen Functions

**Supported graphics drivers
(backends)**
Keyboard Scan Codes
Default Palettes

Runtime Library FAQ
Xbox port FAQ
DOS related FAQ
Windows related FAQ
Linux related FAQ

Miscellaneous

Obsolete Keywords
Glossary
Miscellaneous Keywords
C Standard Library Functions
ASCII Character Codes
Runtime Error Codes
**C/C++ vs. FreeBASIC syntax
comparison**
**C/C++ vs. FreeBASIC integer
data type comparison**

Hacking on FreeBASIC

Developer's Table of Contents

Welcome to FreeBASIC



Welcome to our world! This page is an overview of our online warehouse of knowledge. Enjoy your surfing and we hope this will be the first of many visits.

Introduction

FreeBASIC is a free 32-bit compiler for the BASIC language. It is open source and *licensed under the GPL*. It is designed to be syntax compatible with QuickBASIC, while expanding on the language and capabilities. It can create programs for MS-Windows, DOS and Linux, and is being ported to other platforms. See *About FreeBASIC* and *Main Features*.

Latest Version

FreeBASIC is a beta release compiler and development is ongoing. With each full update, many features are added, and bugs from previous releases are fixed. To see the latest version available, visit <http://sourceforge.net/projects/fbc> on SourceForge, or <http://www.freebasic.net/index.php/download> on FreeBASIC's official website.

Requirements and Installation

Minimum hardware is listed on the *Requirements* page. Visit our *Installation* page for setting up FreeBASIC on your computer.

Running

FreeBASIC is a compiler and as such is not packaged with an IDE (Integrated Development Editor), although there are a few IDE's available. For information on using FreeBASIC without an IDE, see *Running*.

Compatibility with QuickBASIC

FreeBASIC is designed to be syntax compatible with QuickBASIC. For best code-compatibility with QuickBASIC, the *QB dialect* can be used when compiling source code. See *FreeBASIC Dialects* and *Difference from QB*.

Documentation

All official documentation can be found online in the wiki at <http://www.freebasic.net/wiki>. The online documentation is the most up-to-date resource available. In all cases it can be regarded as the correct version. The downloadable versions of the manual are snapshots of the documentation available at a particular time and should be mostly correct for a specific released version of the compiler. However, we do not maintain multiple versions of the documentation so there may be some discrepancies.

Starting points in the Manual

- **Table of Contents**
- **Getting Help with FreeBASIC**
- **Programmer's Guide**

Starting points on the Web

- Official Website at <http://www.freebasic.net>
- Official Forums at <http://www.freebasic.net/forum>
- Official Archive at <http://www.freebasic.net/arch>

Thank you for using FreeBASIC. Happy coding!

There are several options available for getting help with FreeBASIC.

The Manual

This huge user's manual is full of information that can help you learn to write programs using FreeBASIC.

The manual is available online at <http://www.freebasic.net/wiki>. There is search box at the bottom of every page to help you find what you're looking for.

If you are unfamiliar with FreeBASIC or the documentation, you may find these pages a good place to start:

- [Table of Contents](#)
- [Programmer's Guide](#)
- [Library Headers Index](#)
- [Glossary](#)
- [Compiler FAQ](#)
- [Graphics Library FAQ](#)
- [Runtime Library FAQ](#)

A downloadable manual (in CHM format) is available from the sourceforge project page at <http://sourceforge.net/projects/fbc> which features a full table of contents, searching capabilities, an index, plus all the same content as the online version.

Searching the manual on or offline is an excellent place to start finding help about how to write and use FreeBASIC programs.

Examples and Source Code

In the `./examples` directory located where FreeBASIC was installed on your system are hundreds of examples to be compiled and run. Most of the external library examples will need additional libraries to be downloaded to allow them to work. See [Library Headers Index](#) for a full list.

FreeBASIC's official code archive is located at <http://www.freebasic.net/arch>. This archive hosts user contributed libraries and tools and has links to source code located on other websites.

Tutorials

Community created tutorials about FreeBASIC can be found at CommunityTutorials. Some selected tutorials are included in this manual.

FreeBASIC Forum

An active community forum can be found at <http://www.freebasic.net/forum> with several sub-forums. The forum has a search feature that can help you find answers to questions or problems that may have already been asked and solved. First do a search for your problem, if you can't find the answer then post a message in one of the sub-forums.

Chat

IRC or Internet Relay Chat is a great way to chat with the developers and other users, some of whom are very knowledgeable. There are several ways to connect to IRC, if you know what you're doing simply join #freebasic on FreeNode.

If you haven't the foggiest what IRC is and you have Java installed, you can simply go [here](#).

If you're trying to get help, the most important thing is to be patient. Sometimes you won't get a reply right away. Stick around or check back and the Community will try and assist you.

Alphabetical Keywords List



Alphabetical listing of keywords, macros and procedures.

Operators . _ # \$ A B C D E F G H I K L M N O P R S T U V W X Y Z

Operators

- See Operator List

.

- ...

—

- __DATE__
- __Date_Iso__
- __Fb_64Bit__
- __FB_ARGC__
- __FB_ARGV__
- __Fb_Arm__
- __Fb_Asm__
- __Fb_Backend__
- __FB_BIGENDIAN__
- __FB_BUILD_DATE__
- __FB_CYGWIN__
- __FB_DARWIN__
- __FB_DEBUG__
- __FB_DOS__
- __FB_ERR__
- __Fb_Fpmode__
- __Fb_Fpu__
- __FB_FREEBSD__
- __Fb_Gcc__
- __FB_LANG__

K

- Kill

L

- LBound
- LCase
- Left
- Len
- Let
- Lib
- Line
- Line Input
- Line Input #
- LoByte
- LOC
- Local
- Locate
- Lock
- LOF
- Log
- Long
- LongInt
- Loop
- LoWord
- Lpos
- LPrint

- `__FB_LINUX__`
- `__FB_MAIN__`
- `__FB_MIN_VERSION__`
- `__FB_MT__`
- `__FB_NETBSD__`
- `__FB_OPENBSD__`
- `__FB_OPTION_BYVAL__`
- `__FB_OPTION_DYNAMIC__`
- `__FB_OPTION_ESCAPE__`
- `__FB_OPTION_EXPLICIT__`
- `__Fb_Option_Gosub__`
- `__FB_OPTION_PRIVATE__`
- `__FB_OUT_DLL__`
- `__FB_OUT_EXE__`
- `__FB_OUT_LIB__`
- `__FB_OUT_OBJ__`
- `__Fb_Pcos__`
- `__FB_SIGNATURE__`
- `__FB_SSE__`
- `__Fb_Unix__`
- `__Fb_Vectorize__`
- `__FB_VER_MAJOR__`
- `__FB_VER_MINOR__`
- `__FB_VER_PATCH__`
- `__FB_VERSION__`
- `__FB_WIN32__`
- `__FB_XBOX__`
- `__FILE__`
- `__FILE_NQ__`
- `__FUNCTION__`
- `__FUNCTION_NQ__`

M

- LSet
- LTrim
- Mid (Statement)
- Mid (Function)
- Minute
- MKD
- Mkdir
- MKI
- MKL
- MKLongInt
- MKS
- MKShort
- Mod
- Month
- MonthName
- MultiKey
- MutexCreate
- MutexDestroy
- MutexLock
- MutexUnlock

N

- Naked
- Name
- Namespace
- Next
- New
- New (Placement)
- Next (Resume)
- Not
- Now

O

- __LINE__
 - __PATH__
 - __TIME__
- #**
- **#Assert**
 - **#define**
 - **#else**
 - **#elseif**
 - **#endif**
 - **#endmacro**
 - **#error**
 - **#if**
 - **#ifdef**
 - **#ifndef**
 - **#inclid**
 - **#include**
 - **#lang**
 - **#libpath**
 - **#line**
 - **#macro**
 - **#pragma**
 - **#print**
 - **#undef**
- \$**
- **\$Dynamic**
 - **\$Include**
 - **\$Static**
 - **\$Lang**
- A**
- **Abs**
 - **Abstract (Member)**
 - **Access**
 - **Object**
 - **Oct**
 - **OffsetOf**
 - **On Error**
 - **On...Gosub**
 - **On...Goto**
 - **Once**
 - **Open**
 - **Open Com**
 - **Open Cons**
 - **Open Err**
 - **Open Lpt**
 - **Open Pipe**
 - **Open Scrn**
 - **Operator**
 - **Option()**
 - **Option Base**
 - **Option ByVal**
 - **Option Dynamic**
 - **Option Escape**
 - **Option Explicit**
 - **Option Gosub**
 - **Option Nogosub**
 - **Option NoKeyword**
 - **Option Private**
 - **Option Static**
 - **Or**
 - **Or (Graphics Put)**
 - **OrElse**
 - **Out**
 - **Output**

- Acos
- Add (Graphics Put)
- Alias
- Allocate
- Alpha (Graphics Put)
- And
- AndAlso
- And (Graphics Put)
- Any
- Append
- As
- Assert
- AssertWarn
- Asc
- Asin
- Asm
- Atan2
- Atn

B

- Base (Initialization)
- Base (Member Access)
- Beep
- Bin
- Binary
- Bit
- BitReset
- BitSet
- BLoad
- Boolean
- BSave
- Byref (Parameters)
- Byref (Function Results)

P

- Overload
- Override
- Paint
- Palette
- pascal
- PCopy
- Peek
- PMap
- Point
- Pointcoord
- Pointer
- Poke
- Pos
- Preserve
- PReset
- Print
- ?
- Print #
- ? #
- Print Using
- ? Using
- Private
- Private: (Access Control)
- ProcPtr
- Property
- Protected: (Access Control)
- Pset (Statement)
- Pset (Graphics Put)
- Ptr

C

- Byte
- ByVal
- Call
- CAllocate
- Case
- Cast
- Cbool
- CByte
- CDbI
- cdecl
- Chain
- ChDir
- Chr
- CInt
- Circle
- Class
- Clear
- CLng
- CLngInt
- Close
- Cls
- Color
- Command
- Common
- CondBroadcast
- CondCreate
- CondDestroy
- CondSignal
- CondWait
- Const
- Const (Member)

R

- Public
- Public: (Access Control)
- Put (Graphics)
- Put # (File I/O)
- Random
- Randomize
- Read
- Read (File Access)
- Read Write (File Access)
- Reallocate
- ReDim
- Rem
- Reset
- Restore
- Resume
- Resume Next
- Return
- RGB
- RGBA
- Right
- Rmdir
- Rnd
- RSet
- RTrim
- Run
- SAdd
- Scope
- Screen

S

- Const (Qualifier)
- Constructor
- Constructor (Module)
- Continue
- Cos
- CPtr
- CShort
- CSign
- CSng
- CsrLin
- CUByte
- CUInt
- CULng
- CULngInt
- CUnsg
- CurDir
- CUShort
- Custom (Graphics Put)
- CVD
- CVI
- CVL
- CVLongInt
- CVS
- CVShort
- Screen (Console)
- ScreenCopy
- ScreenControl
- ScreenEvent
- ScreenInfo
- ScreenGLProc
- ScreenList
- ScreenLock
- ScreenPtr
- ScreenRes
- ScreenSet
- ScreenSync
- ScreenUnlock
- Second
- Seek (Statement)
- Seek (Function)
- Select Case
- SetDate
- SetEnviron
- SetMouse
- SetTime
- Sgn
- Shared
- Shell

D

- Data
- Date
- DateAdd
- DateDiff
- DatePart
- DateSerial
- DateValue
- Shl
- Shr
- Short
- Sin
- Single
- SizeOf
- Sleep

- Day
- Deallocate
- Declare
- DefByte
- DefDbl
- defined
- DefInt
- DefLng
- Deflongint
- DefShort
- DefSng
- DefStr
- DefUByte
- DefUInt
- Defulongint
- DefUShort
- Delete
- Destructor
- Destructor (Module)
- Dim
- Dir
- Do
- Do...Loop
- Double
- Draw
- Draw String
- DyLibFree
- DyLibLoad
- DyLibSymbol

E

- Else

- Space
- Spc
- Sqr
- Static
- Static (Member)
- stdcall
- Step
- Stick
- Stop
- Str
- Strig
- String (Function)
- String
- StrPtr
- Sub
- Sub (Member)
- Swap
- System

I

- Tab
- Tan
- Then
- This
- Threadcall
- ThreadCreate
- Threaddetach
- ThreadWait
- Time
- TimeSerial
- TimeValue
- Timer
- To

- Elself
- Encoding
- End (Block)
- End (Statement)
- End If
- Enum
- Environ Statement
- Environ
- EOF
- Eqv
- Erase
- Erfn
- Erl
- Ermn
- Err
- Error
- Event (Message Data From
Screenevent)
- Exec
- ExePath
- Exit
- Exp
- Export
- Extends
- Extern
- Extern...End Extern

E

- False
- Field
- FileAttr
- FileCopy
- FileDateTime

- Trans (Graphics Put)
- Trim
- True
- Type (Alias)
- Type (Temporary)
- Type (Udt)
- TypeOf

U

- UBound
- UByte
- UCase
- UInteger
- Ulong
- ULongInt
- Union
- Unlock
- Unsigned
- Until
- UShort
- Using (Print)
- Using (Namespaces)

V

- va_arg
- va_first
- va_next
- Val
- ValNng
- ValInt
- ValUInt
- ValULng
- Var
- VarPtr

- FileExists
- FileLen
- Fix
- Flip
- For
- For...Next
- Format
- Frac
- Fre
- FreeFile
- Function
- Function (Member)

G

- Get (Graphics)
- Get # (File I/O)
- GetJoystick
- GetKey
- GetMouse
- GoSub
- Goto

H

- Hex
- HiByte
- HiWord
- Hour

I

- If...Then
- If
- ImageConvertRow
- ImageCreate
- ImageDestroy
- ImageInfo

W

- View Print
- View (Graphics)
- Virtual (Member)

- Wait
- WBin
- WChr
- Weekday
- WeekdayName
- Wend
- While
- While...Wend
- WHex
- Width
- Window
- WindowTitle
- WInput
- With
- WOct
- Write
- Write #
- Write (File Access)
- WSpace
- WStr
- Wstring (Data Type)
- Wstring (Function)

X

- Xor
- Xor (Graphics Put)

Y

- Year

Z

- Imp
 - Implements
 - Import
 - Inkey
 - Inp
 - Input (Statement)
 - Input (File I/O)
 - Input #
 - Input\$
 - InStr
 - InStrRev
 - Int
 - Integer
 - Is (Select Case)
 - Is (Run-Time Type Information Operator)
 - IsDate
 - Isredirected
- ZString

... (Ellipsis)



Used in place of procedure parameter to pass a variable number of arguments bound in an array declaration to denote that the number of elements will be initialized.

Syntax

```
Declare { Sub | Function } proc_name cdecl ( param_list, ... ) {  
    Dim array_symbol ([lbound To] ...) [As datatype] => { expression  
    #define identifier( [ parameters, ] variadic_parameter... ) body
```

Description

The ellipsis (three dots, ...) is used in procedure declarations and in the variable argument list. A first argument (at least) must always be specified and must be called with the C calling convention `cdecl`. In the procedure body, `va_arg` and `va_next` are used to handle the variable arguments.

Only numeric types and pointers are supported as variable arguments. All variable arguments are implicitly converted to integers, all floating-point variable arguments are implicitly converted to doubles). `Strings` can be passed as a `ZString Ptr` to the string data is taken.

A variadic procedure name can never be overloaded.

Using an ellipsis in place of the upper bound in an array declaration can be set according to the data that appears in the `expression_list`. When used in this manner, an initializer must appear, and cannot be `Any`.

Using an ellipsis behind the last parameter in a `#define` or `#macro` declaration defines a variadic macro. This means it is possible to pass any number of arguments to a variadic macro, which can be used in the `body` as if it was a normal parameter. The `variadic_parameter` will expand to the full list of arguments passed, separated by commas, and can also be completely empty.

Example

```
Declare Function foo cdecl (x As Integer, ...) As
```

```
Dim As Integer myarray(0 To ...) = {0, 1, 2, 3}
Print LBound(myarray), UBound(myarray)    '' 0, 3
```

```
'' Using a variadic macro to wrap a variadic funct
#include "crt.bi"
#define eprintf(Format, args...) fprintf(stderr, F
eprintf(!"Hello from printf: %i %s %i\n", 5, "test

'' LISP-like accessors allowing to modify comma-se
#define car(a, b...) a
#define cdr(a, b...) b
```

Differences from QB

- New to FreeBASIC

See also

- `cdecl`
- `va_arg`
- `va_first`
- `va_next`
- `Dim`
- `Static`
- `#define`

Intrinsic define (macro value) set by the compiler

Syntax

`__DATE__`

Description

Substitutes the compiler date in a literal string ("mm-dd-yyyy" format) where used.

Example

```
Print "Compile Date: " & __DATE__
```

```
Compile Date: 09-29-2011
```

Differences from QB

- New to FreeBASIC

See also

- `__Date_Iso__`
- `__TIME__`
- `Date`

Intrinsic define (macro value) set by the compiler

Syntax

`__DATE_ISO__`

Description

Substitutes the compiler date in a literal string ("yyyy-mm-dd" format) where used. This format is in line with ISO 8601 and can be used for lexicographical date comparisons.

Example

```
Print "Compile Date: " & __DATE_ISO__  
  
If __DATE_ISO__ < "2011-12-25" Then  
    Print "Compiled before Christmas day 2011"  
Else  
    Print "Compiled after Christmas day 2011"  
End If
```

```
Compile Date: 2011-09-29  
Compiled before Christmas day 2011
```

Differences from QB

- New to FreeBASIC

See also

- `__DATE__`

- TIME
- **Date**

Intrinsic define set by the compiler

Syntax

`__FB_64BIT__`

Description

Define created at compile time if the the compilation target is 64bit, otherwise undefined.

Example

```
#ifdef __FB_64BIT__
    '...instructions for 64bit OSes...'
#else
    '...instructions for other OSes'
#endif
```

Differences from QB

- New to FreeBASIC

See also

- `__FB_LINUX__`
- `__FB_FREEBSD__`
- `__FB_OPENBSD__`
- `__FB_NETBSD__`
- `__FB_CYGWIN__`
- `__FB_DARWIN__`
- `__Fb_Pcos__`
- **Compiler Option: -target**

Intrinsic define (macro value) set by the compiler

Syntax

`__FB_ARGC__`

Description

Substituted with the number of arguments passed in on the command

`__FB_ARGC__` is the name of a parameter passed to the program's implicit main function, and therefore is only defined in the module level code of the main module for an application.

Example

```
Dim i As Integer
For i = 0 To __FB_ARGC__ - 1
    Print "arg "; i; " = "; Command(i); ""
Next i
```

Differences from QB

- New to FreeBASIC

See also

- `__FB_ARGV__`
- `Command`

Intrinsic define (macro value) set by the compiler

Syntax

`__FB_ARGV__`

Description

Substituted with a pointer to a list of pointers to the zero terminated command line arguments passed in on the command line.

`__FB_ARGV__` is the name of a parameter passed to the program's implicit main function, and therefore is only defined in the module level code of the main module for an application.

Example

```
Declare Function main _
(
    ByVal argc As Integer, _
    ByVal argv As ZString Ptr Ptr _
) As Integer

End main( __FB_ARGC__, __FB_ARGV__ )

Private Function main _
(
    ByVal argc As Integer, _
    ByVal argv As ZString Ptr Ptr _
) As Integer

    Dim i As Integer
    For i = 0 To argc - 1
        Print "arg "; i; " = "; *argv[i]; ""
    Next i
```

```
Return 0
```

```
End Function
```

Differences from QB

- New to FreeBASIC

See also

- `__FB_ARGC__`
- `Command`

Intrinsic define set by the compiler

Syntax

`__FB_ARM__`

Description

Define created at compile time if the compilation target uses the ARM CPU architecture, otherwise undefined.

Example

```
#ifdef __FB_ARM__
    ...instructions for ARM OSes...
#else
    ...instructions for other OSes
#endif
```

Differences from QB

- New to FreeBASIC

See also

- `__FB_LINUX__`
- `__FB_FREEBSD__`
- `__FB_OPENBSD__`
- `__FB_NETBSD__`
- `__FB_CYGWIN__`
- `__FB_DARWIN__`
- `__Fb_Pcos__`
- **Compiler Option: -target**

Intrinsic define set by the compiler

Syntax

`__FB_ASM__`

Description

`__FB_ASM__` returns a string equal to "intel" or "att" depending on whether inline assembly blocks should use the Intel format or the GCC/AT&T; format.

Example

```
Dim a As Long
#if __FB_ASM__ = "intel"
    Asm
        inc dword Ptr [a]
    End Asm
#else
    Asm
        "incl %0\n" : "+m" (a) : :
    End Asm
#endif
```

Differences from QB

- New to FreeBASIC

See also

- **Compiler Option: -asm**

Intrinsic define set by the compiler

Syntax

`__FB_BACKEND__`

Description

Defined to either "gas" or "gcc", depending on which backend was specified via **-gen**.

Differences from QB

- Did not exist in QB

Intrinsic define set by the compiler

Syntax

`__FB_BIGENDIAN__`

Description

Define without a value created at compile time if compiling for a big endian target.

It can be used to compile parts of the program only if the target is big

Example

```
#ifdef __FB_BIGENDIAN__
    ...instructions only for big endian machines
#else
    ...instructions only for little endian machines
#endif
```

Differences from QB

- Did not exist in QB

__FB_BUILD_DATE__



Intrinsic define (macro string) set by the compiler

Syntax

`__FB_BUILD_DATE__`

Description

Substituted with the quoted string containing the date (MM-DD-YYYY) the

Example

```
Print "This program compiled with a compiler b
```

Differences from QB

- New to FreeBASIC

Intrinsic define set by the compiler

Syntax

`__FB_CYGWIN__`

Description

Define without a value created at compile time in the Cygwin version of the compiler, or when the *-target cygwin* command line option is used. It can be used to compile parts of the program only if the target is Cygwin.

Example

```
#ifdef __FB_CYGWIN__
    ...instructions only for Cygwin...
#else
    ...instructions not for Cygwin...
#endif
```

Differences from QB

- New to FreeBASIC

See also

- `__FB_LINUX__`
- `__Fb_Win32__`
- `__Fb_Unix__`
- **Compiler Option: -target**

Intrinsic define set by the compiler

Syntax

`__FB_DARWIN__`

Description

Define without a value created at compile time in the Darwin version of the compiler, or when the *-target darwin* command line option is used. It can be used to compile parts of the program only if the target is Darwin.

Example

```
#ifdef __FB_DARWIN__
    '...instructions only for Darwin...'
#else
    '...instructions not for Darwin...'
#endif
```

Differences from QB

- New to FreeBASIC

See also

- `__FB_LINUX__`
- `__FB_WIN32__`
- `__Fb_Unix__`
- **Compiler Option: -target**

Intrinsic define (macro value) set by the compiler

Syntax

`__FB_DEBUG__`

Description

`__FB_DEBUG__` indicates if the the generate debug information option '-g' was specified on the command line at the time of compilation.

Returns non-zero (-1) if the option was specified. Returns zero (0) otherwise.

Example

```
#if __FB_DEBUG__ <> 0
    #print Debug mode
#else
    #print Release mode
#endif
```

Differences from QB

- New to FreeBASIC

See also

- `__FB_ERR__`
- `__FB_MT__`
- **Compiler Option: -g**

Intrinsic define set by the compiler

Syntax

__FB_DOS__

Description

Define without a value created at compile time if compiling for the DO target. Default in the DOS hosted version, or active when the **-target dos** command line option is used. It can be used to compile parts of the program only if the target is DOS. Note: the DOS hosted version cannot compile to other targets than DOS by now.

Example

```
#ifdef __FB_DOS__
    ' ... instructions only for DOS ...
    ' ... INT 0x31
#else
    ' ... instructions not for DOS ...
#endif
```

Differences from QB

- New to FreeBASIC

See also

- [__FB_LINUX__](#)
- [__FB_WIN32__](#)
- [__Fb_Pcos__](#)
- [DOS related FAQ](#)
- [Compiler Option: -target](#)

Intrinsic define (macro value) set by the compiler

Syntax

`__FB_ERR__`

Description

`__FB_ERR__` indicates if **-e**, **-ex**, or **-exx** was specified on the compiler compilation of a module.

Returns one of the following values:

value	description
0	'-e', '-ex', '-exx' not specified
1	'-e' was specified
3	'-ex' was specified
7	'-exx' was specified

`__FB_ERR__` is always defined.

Example

```
'Example code to demonstrate a use of __FB_ERR__
Dim err_command_line As UByte
err_command_line = __FB_ERR__
Select Case err_command_line
Case 0
Print "No Error Checking enabled on the Command Li
Case 1
Print "Some Error Checking enabled on the Command
Case 3
Print "QBasic style Error Checking enabled on the
Case 7
Print "Extreme Error Checking enabled on the Comma
```

```
Case Else  
Print "Some Unknown Error level has been set!"  
End Select
```

Differences from QB

- New to FreeBASIC

See also

- `__FB_MT__`
- `__FB_DEBUG__`
- **Compiler Option: -e**
- **Compiler Option: -ex**
- **Compiler Option: -exx**
- **Error Handling**

Intrinsic define set by the compiler

Syntax

`__FB_FPMODE__`

Description

Defined as "fast" if SSE fast arithmetics is enabled, or "precise" otherwise.

Example

```
#if __FB_FPMODE__ = "fast"
  ' ... instructions for using fast-
  mode math ...
#else
  ' ... instructions for using normal math ...
#endif
```

Differences from QB

- New to FreeBASIC

See also

- [Compiler Option: -fpmode](#)

Intrinsic define set by the compiler

Syntax

`__FB_FPU__`

Description

Defined as "sse" if SSE floating point arithmetics is enabled, or "x87" otherwise.

Example

```
#if __FB_FPU__ = "sse"  
  ' ... instructions only for SSE ...  
#else  
  ' ... instructions not for SSE ...  
#endif
```

Differences from QB

- New to FreeBASIC

See also

- [__FB_SSE__](#)
- **Compiler Option: -fpu**

Intrinsic define set by the compiler

Syntax

`__FB_FREEBSD__`

Description

Define without a value created at compile time in the FreeBSD version of the compiler, or when the *-target freebsd* command line option is used. It can be used to compile parts of the program only if the target is FreeBSD.

Example

```
#ifdef __FB_FREEBSD__
    ...instructions only for FreeBSD...
#else
    ...instructions not for FreeBSD...
#endif
```

Differences from QB

- New to FreeBASIC

See also

- `__FB_LINUX__`
- `__FB_WIN32__`
- `__Fb_Unix__`
- **Compiler Option: -target**

Intrinsic define set by the compiler

Syntax

`__FB_GCC__`

Description

Defined to true (-1) if **-gen gcc** is used, or false (0) otherwise.

Differences from QB

- Did not exist in QB

Intrinsic define (macro value) set by the compiler

Syntax

`__FB_LANG__`

Description

`__FB_LANG__` indicates which language compatibility option was set at the time of compilation of a module. By default `__FB_LANG__` will be set to "fb". The language compatibility option can be changed using one (or more) of the following methods:

- **-lang** command line option
- **-forcelang** command line option
- **#lang** directive
- **\$Lang** metacommand

Returns a lower case string with one of the following values:

value	description
"fb"	FreeBASIC compatibility (default)
"qb"	QBASIC compatibility
"fblite"	FreeBASIC language compatibility, with a more QBASIC-compatible coding style
"deprecated"	FBC version 0.16 compatibility

`__FB_LANG__` is always defined.

Example

```
' ' Set option explicit always on  
  
#ifdef __FB_LANG__  
  #if __FB_LANG__ <> "fb"
```

```
    Option Explicit
#endif
#else
'' Older version - before lang fb
    Option Explicit
#endif
```

Differences from QB

- New to FreeBASIC

See also

- `__FB_VERSION__`
- `#lang`
- **Compiler Option: -lang**
- **Compiler Option: -forcelang**
- **Compiler Dialects**

Intrinsic define set by the compiler

Syntax

__FB_LINUX__

Description

Define without a value created at compile time when compiling to the Linux target. Default in the Linux hosted version of the compiler, or active when the *-target linux* command line option is used. It can be used to compile parts of the program only if the target is Linux.

Example

```
#ifdef __FB_LINUX__
' ... instructions only for Linux ...
' ... #libpath "/usr/X11/lib"
#else
' ... instructions not for Linux ...
#endif
```

Differences from QB

- New to FreeBASIC

See also

- [__FB_DOS__](#)
- [__FB_WIN32__](#)
- [__Fb_Unix__](#)
- **Compiler Option: -target**

Intrinsic define set by the compiler

Syntax

`__FB_MAIN__`

Description

`__FB_MAIN__` is defined in the main module and not defined in other modules.

The main module is determined by the compiler as either the first source file listed on the command line or explicitly named using the `-n` option on the command line.

Example

```
#ifdef __FB_MAIN__
    #print Compiling the main module
#else
    #print Compiling an additional module
#endif
```

Differences from QB

- New to FreeBASIC

See also

- **Compiler Option: `-m`**
- `#ifdef`
- `#ifndef`

Macro function to test minimum compiler version

Syntax

```
#define __FB_MIN_VERSION__( major, minor, patch) _  
(( __FB_VER_MAJOR__ > major) or _  
(( __FB_VER_MAJOR__ = major) and (( __FB_VER_MINOR__ > minor) or _  
(__FB_VER_MINOR__ = minor and __FB_VER_PATCH__ >= patch_level)))
```

Usage

```
__FB_MIN_VERSION__( major, minor, patch)
```

Parameters

major
minimum major version to test
minor
minimum minor version to test
patch
minimum patch version to test

Return Value

Returns zero (0) if the compiler version is less than the specified version
version is greater than or equal to specified version

Description

__FB_MIN_VERSION__ tests for a minimum version of the compiler.

Example

```
#if Not __FB_MIN_VERSION__(0, 18, 2)  
    #error fbc must be at least version 0.18.2 To  
#endif
```

Differences from QB

- New to FreeBASIC

See also

- [#if](#)

Intrinsic define (macro value) set by the compiler

Syntax

`__FB_MT__`

Description

`__FB_MT__` indicates if the the multithreaded option ***-mt*** was specified on the command line at the time of compilation.

Returns non-zero (-1) if the option was specified. Returns zero (0) otherwise.

Example

```
#if __FB_MT__
    #print Using multi-threaded library
#else
    #print Using Single-threaded library
#endif
```

Differences from QB

- New to FreeBASIC

See also

- `__FB_DEBUG__`
- **Compiler Option: -mt**

Intrinsic define set by the compiler

Syntax

`__FB_NETBSD__`

Description

Define without a value created at compile time in the NetBSD version of the compiler, or when the *-target netbsd* command line option is used. It can be used to compile parts of the program only if the target is NetBSD.

Example

```
#ifdef __FB_NETBSD__
    '...instructions only for NetBSD...'
#else
    '...instructions not for NetBSD...'
#endif
```

Differences from QB

- New to FreeBASIC

See also

- `__FB_LINUX__`
- `__FB_WIN32__`
- `__Fb_Unix__`
- **Compiler Option: -target**

Intrinsic define set by the compiler

Syntax

`__FB_OPENBSD__`

Description

Define without a value created at compile time in the OpenBSD version of the compiler, or when the *-target openbsd* command line option is used. It can be used to compile parts of the program only if the target is OpenBSD.

Example

```
#ifdef __FB_OPENBSD__
    ...instructions only for OpenBSD...
#else
    ...instructions not for OpenBSD...
#endif
```

Differences from QB

- New to FreeBASIC

See also

- `__FB_LINUX__`
- `__FB_WIN32__`
- `__Fb_Unix__`
- **Compiler Option: -target**

Intrinsic define (macro value) set by the compiler

Syntax

`__FB_OPTION_BYVAL__`

Description

Indicates if parameters to a **Function** or **Sub** are passed by reference or by value as with **ByVal** by default when the by value / by reference is explicitly stated.

`__FB_OPTION_BYVAL__` is set to non-zero (-1) if by default parameters are by value, and zero (0) if by default parameters are passed by reference.

The default for passing parameters by reference or by value is determined by the **lang** command line option used during compilation or usage of **option** source file.

Example

```
#if( __FB_OPTION_BYVAL__ <> 0 )
  #error Option ByVal must Not be used With This s
#endif
```

Differences from QB

- New to FreeBASIC

See also

- **ByVal**
- **ByRef**
- **Option ByVal**

Intrinsic define (macro value) set by the compiler

Syntax

`__FB_OPTION_DYNAMIC__`

Description

`__FB_OPTION_DYNAMIC__` is defined as true (negative one (-1)) if a recent **Option Dynamic** statement or '**\$Dynamic**' meta-command was issued. Otherwise, it is defined as zero (0).

Example

```
#if __FB_OPTION_DYNAMIC__ <> 0
#error This module must Not use Option Dynamic
#endif
```

Differences from QB

- New to FreeBASIC

See also

- **Option Dynamic**
- **Option Static**

__FB_OPTION_ESCAPE__



Intrinsic define (macro value) set by the compiler

Syntax

`__FB_OPTION_ESCAPE__`

Description

Indicates if by default, string literals are processed for escape charact prefixed with the **\$ Operator** for non-escaped strings, or the **! Operat**

The default method for processing string literals is set by usage of the option during compilation or usage of **option Escape** in the source file.

`__FB_OPTION_ESCAPE__` returns zero (0) if the option has not been set. F the option has been set.

Example

```
#if( __FB_OPTION_ESCAPE__ <> 0 )
  #error Option Escape must Not be used With This
#endif
```

Differences from QB

- New to FreeBASIC

See also

- **Option Escape**

Intrinsic define (macro value) set by the compiler

Syntax

`__FB_OPTION_EXPLICIT__`

Description

`__FB_OPTION_EXPLICIT__` indicates if **Option Explicit** has been used p in the source.

Returns zero (0) if the option has not been set. Returns non-zero (-1) option has been set.

Example

```
#if( __FB_OPTION_EXPLICIT__ = 0 )  
  #error Option Explicit must used With This modul  
#endif
```

Differences from QB

- New to FreeBASIC

See also

- **Dim**
- **Option Explicit**

__Fb_Option_Gosub__



Intrinsic define (macro value) set by the compiler

Syntax

`__FB_OPTION_GOSUB__`

Description

Indicates how **GoSub** and **Return** will be handled at compile time. If the option is set (-1) then **GoSub** is allowed and **Return** is recognized as return-from-gosub only. If the option is not set (0) then **GoSub** is not allowed and **Return** is recognized as return-from-procedure only.

This macro value can be changed at compile time. **Option Gosub** will set the option (enable gosub support) and **Option Nogosub** will clear the option (disable gosub support).

`__FB_OPTION_GOSUB__` returns zero (0) if the option has not been set. Returns non-zero (-1) if the option has been set.

Example

```
#if( __FB_OPTION_GOSUB__ <> 0 )
    ' turn off gosub support
    Option nogosub
#endif
```

Dialect Differences

- Defaults to -1 in the *-lang qb* dialect and 0 in all other dialects.

Differences from QB

- New to FreeBASIC

See also

- **Option Gosub**
- **Option Nogosub**

__FB_OPTION_PRIVATE__



Intrinsic define (macro value) set by the compiler

Syntax

`__FB_OPTION_PRIVATE__`

Description

Indicates if by default **Function**'s and **Sub**'s have module scope or global scope not explicitly specified with **Private** or **Public**.

The default scope specifier for functions and subs is set by usage of the command line option during compilation or usage of **Option Private** in the source code.

`__FB_OPTION_PRIVATE__` returns zero (0) if the option has not been set. zero (-1) if the option has been set.

Example

```
#if( __FB_OPTION_PRIVATE__ <> 0 )
  #error Option Private must Not be used With This
#endif
```

Differences from QB

- New to FreeBASIC

See also

- **Option Private**
- **Private**
- **Public**

Intrinsic define (macro value) set by the compiler

Syntax

`__FB_OUT_DLL__`

Description

`__FB_OUT_DLL__` indicates that the specified output file type on the compilation is a shared library.

Returns non-zero (-1) if the output is a shared library. Returns zero (0) otherwise.

Only one of `__FB_OUT_DLL__`, `__FB_OUT_EXE__`, `__FB_OUT_LIB__`, or `__FB_OUT_OBJ__` will evaluate to non-zero (-1). All others will evaluate to zero (0).

Example

```
#if __FB_OUT_DLL__
    ... specific instructions when making a shared library
#else
    ... specific instructions when not making a shared library
#endif
```

Differences from QB

- New to FreeBASIC

See also

- `__FB_OUT_EXE__`
- `__FB_OUT_LIB__`
- `__FB_OUT_OBJ__`
- **Compiler Option: -dll**

Intrinsic define (macro value) set by the compiler

Syntax

`__FB_OUT_EXE__`

Description

`__FB_OUT_EXE__` indicates that the specified output file type on the command line of compilation is an executable.

Returns non-zero (-1) if the output is an executable. Returns zero (0) otherwise.

Only one of `__FB_OUT_DLL__`, `__FB_OUT_EXE__`, `__FB_OUT_LIB__`, or `__FB_OUT_OBJ__` will be non-zero (-1). All others will evaluate to zero (0).

Example

```
#if __FB_OUT_EXE__
    ... specific instructions when making an executable
#else
    ... specific instructions when not making an executable
#endif
```

Differences from QB

- New to FreeBASIC

See also

- `__FB_OUT_DLL__`
- `__FB_OUT_LIB__`
- `__FB_OUT_OBJ__`

Intrinsic define (macro value) set by the compiler

Syntax

`__FB_OUT_LIB__`

Description

`__FB_OUT_LIB__` indicates that the specified output file type on the command line of compilation is a static library.

Returns non-zero (-1) if the output is a static library. Returns zero (0) otherwise.

Only one of `__FB_OUT_DLL__`, `__FB_OUT_EXE__`, `__FB_OUT_LIB__`, or `__FB_OUT_OBJ__` is set to non-zero (-1). All others will evaluate to zero (0).

Example

```
#if __FB_OUT_LIB__
    ... specific instructions when making a static library
#else
    ... specific instructions when not making a static library
#endif
```

Differences from QB

- New to FreeBASIC

See also

- `__FB_OUT_EXE__`
- `__FB_OUT_DLL__`
- `__FB_OUT_OBJ__`
- **Compiler Option: `-lib`**

Intrinsic define (macro value) set by the compiler

Syntax

`__FB_OUT_OBJ__`

Description

`__FB_OUT_OBJ__` indicates that the specified output file type on the compilation is an object module.

Returns non-zero (-1) if the output is an object module. Returns zero (0) otherwise.

Only one of `__FB_OUT_DLL__`, `__FB_OUT_EXE__`, `__FB_OUT_LIB__`, or `__FB_OUT_OBJ__` is defined. All others will evaluate to zero (0).

Example

```
#if __FB_OUT_OBJ__
    ... specific instructions when compiling
#else
    ... specific instructions when not compiling
#endif
```

Differences from QB

- New to FreeBASIC

See also

- `__FB_OUT_EXE__`
- `__FB_OUT_DLL__`
- `__FB_OUT_LIB__`

Intrinsic define set by the compiler

Syntax

`__FB_PCOS__`

Description

Define created at compile time if the OS has filesystem behavior style like common PC OSes, e.g. DOS, Windows, OS/2, Symbian OS, possibly others. Drive letters, backslashes, that stuff, otherwise undefined.

Example

```
#ifdef __FB_PCOS__
    ...instructions for PC-ish OSes...
#else
    ...instructions for other OSes
#endif
```

Differences from QB

- New to FreeBASIC

See also

- `__FB_WIN32__`
- `__FB_DOS__`
- `__FB_XBOX__`
- `__Fb_Unix__`
- **Compiler Option: -target**

Intrinsic define (macro string) set by the compiler

Syntax

```
__FB_SIGNATURE__
```

Description

Substituted by a signature of the compiler where used.

Example

```
Print __FB_SIGNATURE__
```

```
FreeBASIC 0.21.1
```

Differences from QB

- New to FreeBASIC

See also

- [__FB_VERSION__](#)
- [__FB_WIN32__](#)
- [__FB_LINUX__](#)
- [__FB_DOS__](#)

Intrinsic define set by the compiler

Syntax

`__FB_SSE__`

Description

Define without a value created at compile time if SSE floating point arithmetics is enabled.

Example

```
#ifdef __FB_SSE__
' ... instructions only for SSE ...
#else
' ... instructions not for SSE ...
#endif
```

Differences from QB

- New to FreeBASIC

See also

- [__Fb_Fpu__](#)
- **Compiler Option: -fpu**

Intrinsic define set by the compiler

Syntax

`__FB_UNIX__`

Description

Define created at compile time if the OS is reasonably enough like UNIX that you can call it UNIX, otherwise undefined.

Example

```
#ifdef __FB_UNIX__
    ...instructions for UNIX-family OSes...
#else
    ...instructions for other OSes
#endif
```

Differences from QB

- New to FreeBASIC

See also

- `__FB_LINUX__`
- `__FB_FREEBSD__`
- `__FB_OPENBSD__`
- `__FB_NETBSD__`
- `__FB_CYGWIN__`
- `__FB_DARWIN__`
- `__Fb_Pcos__`
- **Compiler Option: -target**

Intrinsic define set by the compiler

Syntax

`__FB_VECTORIZE__`

Description

Defined as the vectorisation level number set by the **-vec** command-li

Example

```
#if __FB_VECTORIZE__ = 2
    ' ... instructions only for vectorization level
#else
    ' ...
#endif
```

Differences from QB

- New to FreeBASIC

See also

- **Compiler Option: -vec**

Intrinsic define (macro value) set by the compiler

Syntax

`__FB_VER_MAJOR__`

Description

`__FB_VER_MAJOR__` will return the major version of FreeBASIC currently 0.90, and will remain 0 until FreeBASIC version 1.0 is released.

Example

```
Dim fbMajorVersion As Integer
Dim fbMinorVersion As Integer
Dim fbPatchVersion As Integer

fbMajorVersion = __FB_VER_MAJOR__
fbMinorVersion = __FB_VER_MINOR__
fbPatchVersion = __FB_VER_PATCH__

Print "Welcome to FreeBASIC " & fbMajorVersion & "
```

Differences from QB

- New to FreeBASIC

See also

- `__FB_VER_MINOR__`
- `__FB_VER_PATCH__`

Intrinsic define (macro value) set by the compiler

Syntax

`__FB_VER_MINOR__`

Description

`__FB_VER_MINOR__` will return the minor version of FreeBASIC currently
minor version number is 90.

Example

```
Dim fbMajorVersion As Integer
Dim fbMinorVersion As Integer
Dim fbPatchVersion As Integer

fbMajorVersion = __FB_VER_MAJOR__
fbMinorVersion = __FB_VER_MINOR__
fbPatchVersion = __FB_VER_PATCH__

Print "Welcome to FreeBASIC " & fbMajorVersion & "
```

Differences from QB

- New to FreeBASIC

See also

- `__FB_VER_MAJOR__`
- `__FB_VER_PATCH__`

Intrinsic define (macro value) set by the compiler

Syntax

`__FB_VER_PATCH__`

Description

`__FB_VER_PATCH__` will return the patch/subversion/revision number the example, there were subversions 1, 2, 3, 4, 5 and 6, resulting in versio

Example

```
Dim fbMajorVersion As Integer
Dim fbMinorVersion As Integer
Dim fbPatchVersion As Integer

fbMajorVersion = __FB_VER_MAJOR__
fbMinorVersion = __FB_VER_MINOR__
fbPatchVersion = __FB_VER_PATCH__

Print "Welcome to FreeBASIC " & fbMajorVersion & "
```

Differences from QB

- New to FreeBASIC

See also

- `__FB_VER_MAJOR__`
- `__FB_VER_MINOR__`

Intrinsic define (macro string) set by the compiler

Syntax

`__FB_VERSION__`

Description

Substituted by the version number of the compiler where used.

Example

```
#if __FB_VERSION__ < "0.18"  
#error Please compile With FB version 0.18 Or abc  
#endif
```

This will stop the compilation if the compiler version is below 0.18

Differences from QB

- Did not exist in QB

See also

- `__FB_SIGNATURE__`
- `__FB_WIN32__`
- `__FB_LINUX__`
- `__FB_DOS__`

Intrinsic define set by the compiler

Syntax

`__FB_WIN32__`

Description

Define without a value created at compile time if compiling to the Win32 target. Default in Win32 hosted version, or active if the ***-target win32*** command line option is used. It can be used to compile parts of the program only if the target is Win32.

Example

```
#ifdef __FB_WIN32__
' ... instructions only for Win32 ...
' ... GetProcAddress ...
#else
' ... instructions not for Win32 ...
#endif
```

Differences from QB

- New to FreeBASIC

See also

- [__FB_DOS__](#)
- [__FB_LINUX__](#)
- [__Fb_Pcos__](#)
- **Compiler Option: -target**

Intrinsic define set by the compiler

Syntax

`__FB_XBOX__`

Description

Define without a value created at compile time when the *-target xbox* command line option is used. It can be used to compile parts of the program only if the target is Xbox.

Example

```
#ifdef __FB_XBOX__
    '...instructions only for Xbox...'
#else
    '...instructions not for Xbox...'
#endif
```

Differences from QB

- New to FreeBASIC

See also

- `__FB_LINUX__`
- `__FB_WIN32__`
- **Compiler Option: -target**

Intrinsic define (macro string) set by the compiler

Syntax

`__FILE__`

Description

Substituted with the quoted source file name where used.

An example of normal use is to report wrong values in debugging.

Example

```
Dim a As Integer
If a<0 Then
    Print "Error: a = " & a & " in " & __FILE__ &
End If
```

```
Error: a = -32767 in test.bas (MAIN) line 47
```

Differences from QB

- Did not exist in QB

See also

- `__FILE_NQ__`
- `__FUNCTION__`
- `__LINE__`

Intrinsic define (macro string) set by the compiler

Syntax

`__FILE_NQ__`

Description

Substituted with the non-quoted source file name where used.

Example

```
#print __FILE_NQ__
```

Differences from QB

- New to FreeBASIC

See also

- `__FILE__`
- `__FUNCTION_NQ__`
- `__LINE__`

__FUNCTION__



Intrinsic define (macro string) set by the compiler

Syntax

`__FUNCTION__`

Description

Substituted with the quoted name of the current function block where
Its normal use is to report wrong values in debugging.

If `__FUNCTION__` is used at the module level, the function name given with
`__FB_MODLEVELPROC__` for a different module.

Example

```
Dim a As Integer
' ...

If a < 0 Then ' this shouldn't happen
    Print "Error: a = " & a & " in " & __FILE__ &
End If
```

```
Error: a = -32767 in test.bas (__FB_MAINPROC__) line 47
```

Differences from QB

- Did not exist in QB

See also

- FILE
- FUNCTION_NQ
- LINE

__FUNCTION_NQ__



Intrinsic define (macro string) set by the compiler

Syntax

`__FUNCTION_NQ__`

Description

Substituted with the non-quoted name of the current function block where used.

If `__FUNCTION_NQ__` is used at the module level, the function name given will be `__FB_MAINPROC__` for the main module, or `__FB_MODLEVELPROC__` if a different module. This is not the actual function name though, so it's not as useful there.

Example

```
Sub MySub
  Print "Address of " + __FUNCTION__ + " is ";
  Print Hex( @__FUNCTION_NQ__ )
End Sub
```

MySub

```
Address of MYSUB is 4012D0
```

Differences from QB

- Did not exist in QB

See also

- FILE_NQ
- FUNCTION
- LINE

Intrinsic define (macro value) set by the compiler

Syntax

`__LINE__`

Description

Substituted with the current line number of the source file where used

Its normal use is to report wrong values in debugging.

Example

```
Dim a As Integer

If a < 0 Then
    Print "Error: a = " & a & " in " & __FILE__ &
End If
```

```
Error: a = -32767 in test.bas (MAIN) line 47
```

Differences from QB

- Did not exist in QB

See also

- `__FILE__`
- `__FUNCTION__`

Intrinsic define (macro string) set by the compiler

Syntax

`__PATH__`

Description

Set to the quoted absolute path of the source file at the time of compilation.

Example

```
' Tell the compiler to search the source file's  
' directory for libraries  
  
#libpath __PATH__
```

Differences from QB

- New to FreeBASIC

See also

- [__FILE__](#)

Intrinsic define (macro value) set by the compiler

Syntax

`__TIME__`

Description

Substitutes the compiler time in a literal string (24 clock, "hh:mm:ss" format) where used.

Example

```
Print "Compile Time: " & __TIME__
```

```
Compile Time: 13:42:57
```

Differences from QB

- New to FreeBASIC

See also

- `__DATE__`
- `__Date_Iso__`
- `Time`

Preprocessor conditional directive

Syntax

```
#assert condition
```

Parameters

condition

A conditional expression that is assumed to be true

Description

Asserts the truth of a conditional expression at compile time. If *condition* is false, compilation will stop with an error.

This statement differs from the **Assert** macro in that **#assert** is evaluated at compile-time and **Assert** is evaluated at run-time.

Example

```
Const MIN = 5, MAX = 10
#assert MAX > MIN ' cause a compile-
time error if MAX <= MIN
```

Differences from QB

- New to FreeBASIC

See also

- **Assert**
- **#if**
- **#error**

Preprocessor directive to define a macro

Syntax

```
#define identifier body  
#define identifier( [ parameters ] ) body  
#define identifier( [ parameters, ] Variadic_Parameter... ) body
```

Description

#define allows to declare text-based preprocessor macros. Once the **#define**, it will start replacing further occurrences of *identifier* with *body*. The expansion is done recursively, until there is nothing more to expand. **#undef** can be used to make the **#define**.

Parameters turn a define into a function-like macro, allowing text arguments. Any occurrences of the parameter names in the *body* will be replaced by the argument text during expansion. The **# Stringize** operator can be used to turn them into string literals, and the **## Concatenate** operator can be used to concatenate them together.

Note: In the function-like macro declaration, the *identifier* should be enclosed in parentheses (()) immediately without any white-space in between, otherwise the compiler will treat it as part of the *body*.

Defines are *scoped*; they are only visible in the scope they were defined in. If the *identifier* is not enclosed in parentheses, the define is visible throughout the module. If the *identifier* is enclosed in parentheses, the define is only visible within that scope. Namespaces on the other hand have no effect on the visibility of a define.

Identifiers can be checked for with **#ifdef** and others, which can be used for conditional compiling from the compiler.

The result of macro expansion can be checked by using the **-pp** compiler option.

`#defines` are often used to declare constants. The `const` statement is

Example

```
' ' Definition and check
#define DEBUGGING
#ifdef DEBUGGING
    ' ... statements
#endif

' ' Simple definition/text replacement
#define FALSE 0
#define TRUE (Not FALSE)

' ' Function-like definition
#define MyRGB(R,G,B) (((R)Shl 16) Or ((G)Shl 8) C
Print Hex( MyRGB(&hff, &h00, &hff) )

' ' Line continuation and statements in a definitio
#define printval(bar) _
    Print #bar; " ="; bar

' ' #defines are visible only in the scope where th
Scope
    #define LOCALDEF 1
End Scope

#ifdef LOCALDEF
#    Print LOCALDEF Is Not defined
#endif

' ' namespaces have no effect on the visibility of
Namespace foo
#    define NSDEF
End Namespace

#ifdef NSDEF
#    Print NSDEF Is defined
```

```
#endif
```

Differences from QB

- New to FreeBASIC

See also

- `#macro`
- `# Preprocessor Stringize`
- `## Preprocessor Concatenate`
- `#ifdef`
- `#undef`
- `Const`
- ...

Preprocessor conditional directive

Syntax

```
#if (expression)
' Conditionally included statements if expression is True
#else
' Conditionally included statements if expression is False
#endif
```

Description

#else can be added to an **#if**, **#ifdef**, or **#ifndef** block to provide an alternate result to the conditional expression.

Example

```
#define MODULE_VERSION 1
Dim a As String
#if (MODULE_VERSION > 0)
    a = "Release"
#else
    a = "Beta"
#endif
Print "Program is "; a
```

Differences from QB

- New to FreeBASIC

See also

- **#define**
- **#macro**
- **#if**

- `#elseif`
- `#endif`
- `#ifdef`
- `#ifndef`
- `#undef`
- `defined`

Preprocessor conditional directive

Syntax

```
#if (expression1)
' Conditionally included statements if expression1 is True
#elseif (expression2)
' Conditionally included statements if expression2 is True
#else
' Conditionally included statements if both
' expression1 and expression2 are False
#endif
```

Description

#elseif can be added to an **#if** block to provide an additional conditions.

Example

```
#define WORDSIZE 16
#if (WORDSIZE = 16)
' Do some some 16 bit stuff
#elseif (WORDSIZE = 32)
' Do some some 32 bit stuff
#else
#error WORDSIZE must be set To 16 Or 32
#endif
```

Differences from QB

- New to Freebasic

See also

- **#define**

- `#macro`
- `#if`
- `#else`
- `#endif`
- `#ifdef`
- `#ifndef`
- `#undef`
- `defined`

Preprocessor conditional directive

Syntax

```
#endif
```

Description

Ends a group of conditional directives

See `#if`, `#ifdef`, or `#ifndef` for examples of usage.

Example

```
#define DEBUG_LEVEL 1
#if (DEBUG_LEVEL = 1)
    'Conditional statements
#endif
```

Differences from QB

- New to FreeBASIC

See also

- `#define`
- `#macro`
- `#if`
- `#else`
- `#elseif`
- `#ifdef`
- `#ifndef`
- `#undef`

- `defined`

#Macro...#Endmacro



Preprocessor directive to define a multiline macro

Syntax

```
#macro identifier( [ parameters ] )  
body  
#endmacro
```

```
#macro identifier( [ parameters, ] Variadic_Parameter... )  
body  
#endmacro
```

Description

#macro is the multi-line version of #define.

Example

```
' ' macro as an expression value  
#macro Print1( a, b )  
    a + b  
#endmacro  
  
Print Print1( "Hello", "World" )  
  
' ' Output :  
' ' Hello World!
```

```
' ' macro as multiple statements  
#macro Print2( a, b )  
    Print a;  
    Print " ";  
    Print b;  
    Print "!"  
#endmacro
```

```
Print2( "Hello", "World" )
```

```
' ' Output :  
' ' Hello World!
```

Differences from QB

- New to FreeBASIC

See also

- `#define`
- `#ifdef`
- `#undef`

#error



Preprocessor diagnostic directive

Syntax

```
#error error_text
```

Parameters

error_text

The display message

Description

#error stops compiling and displays *error_text* when compiler finds it

This keyword must be surrounded by an **#if** *<condition>* ...**#endif**, so the compiler can reach **#error** only if *<condition>* is met.

Example

```
#define c 1

#if c = 1
    #error Bad value of c
#endif
```

Differences from QB

- New to FreeBASIC

See also

- **#if**
- **#print**

- #Assert

Preprocessor conditional directive

Syntax

```
#if (expression)
' Conditionally included statements
#endif
```

Description

Conditionally includes statements at compile time.

Statements contained within the `#if` / `#endif` block are included if *expr* True (non-zero) and excluded (ignored) if *expression* evaluates to False.

This conditional directive differs from the `if` conditional statement in that it is evaluated at compile-time and `if` is evaluated at run-time.

Example

```
#define DEBUG_LEVEL 1
#if (DEBUG_LEVEL >= 2)
' This line is not compiled since the expression is false
Print "Starting application"
#endif
```

Differences from QB

- New to FreeBASIC

See also

- `#define`
- `#macro`

- `#else`
- `#elseif`
- `#endif`
- `#ifdef`
- `#ifndef`
- `#undef`
- `defined`
- `#Assert`

Preprocessor conditional directive

Syntax

```
#ifdef symbol
' Conditionally included statements
#endif
```

Description

Conditionally includes statements at compile time.

Statements within the `#ifdef...#endif` block are included if *symbol* is defined and excluded (ignored) if *symbol* is not defined.

`#ifdef symbol` is equivalent to `#if defined (symbol)`

Example

```
#define _DEBUG
#ifdef _DEBUG
    ' Special statements for debugging
#endif
```

Differences from QB

- New to Freebasic

See also

- `#define`
- `#macro`
- `#if`
- `#else`

- `#elseif`
- `#endif`
- `#ifndef`
- `#undef`
- `defined`

Preprocessor conditional directive

Syntax

```
#ifndef symbol
' Conditionally included statements
#endif
```

Description

Conditionally includes statements at compile time.

Statements within the `#ifndef...#endif` block are included if *symbol* is not defined and excluded (ignored) if *symbol* is defined.

`#ifndef symbol` is equivalent to `#if Not defined(symbol)`

Example

```
#ifndef __MYFILE_BI__
#define __MYFILE_BI__
    ' Declarations
#endif
```

Differences from QB

- New to FreeBASIC

See also

- `#define`
- `#macro`
- `#if`
- `#else`

- `#elseif`
- `#endif`
- `#ifdef`
- `#undef`
- `defined`

Preprocessor directive

Syntax

```
#includ "libname"
```

Description

Includes a library in the linking process as if the user specified `-l libname` command line.

Example

```
' ' incomplete code snippet  
  
' ' this will include libmystuff.a in the link process  
#includ "mystuff"
```

Differences from QB

- New to FreeBASIC

See also

- `#include`
- **Compiler Option: -l**
- **Compiler Option: -p**

Preprocessor statement to include contents of another source file

Syntax

```
#include [once] "file"
```

Description

`#include` inserts source code from another file at the point where the `#include` directive appears. This has the effect of compiling the source code from the include file as though it were part of the source file that includes it. Once the compiler has reached the end of the include file, the original source file continues to be compiled.

This is useful to remove code from a file and separate it into more files. It is useful to have a single file with declarations in a program formed by several modules. You may include files within an include file, although avoid including the original file into itself, this will not produce valid results. Typically, include files will have an extension of `.bi` and are mainly used for declaring subs/functions/variables of a library, but any valid source code may be present in an include file.

The `once` specifier tells the compiler to include the file only once even if it is included several times by the source code.

`$Include` is an alternative form of `include`, existing only for compatibility with QuickBASIC. It is recommended to use `#include` instead.

The compiler will automatically convert path separator characters (`'/'` and `'\'`) as needed to properly load the file. The filename name may be an absolute or relative path.

For relative paths, or where no path is given at all, the include file is searched for in the following order:

- Relative from the directory of the source file
- Relative from the current working directory

- Relative from addition directories specified with the `-i` command line option
- The include folder of the FreeBASIC installation (`FreeBASIC\include` where `FreeBASIC` is the folder where the `fbcc` executable is located)

Example

```
' header.bi file
Type FooType
    Bar As Byte
    Barbeque As Byte
End Type
```

```
' main.bas file
#include "header.bi"

Dim Foo As FooType

Foo.Bar = 1
Foo.Barbeque = 2
```

Differences from QB

- New to FreeBASIC

See also

- `#define`
- `#includelib`
- **Compiler Option: `-i`**
- **Compiler Option: `-include`**

Preprocessor statement to set the compiler dialect.

Syntax

```
#lang "lang"
```

Parameters

"lang"

The dialect to set, enclosed in double quotes, and must be one of "fb", "fbLite", "qb", or "deprecated".

Description

If the *-forcelang* option was not given on the command line, `#lang` can be used to set the dialect for the source module in which it appears. At most two passes will be made on the source module. On the first pass, if the specified dialect is anything other than the default dialect (chosen with *-lang*, or "fb" by default), the compiler will reset the parser for another pass and restart compilation at the beginning of the source module. If this directive is encountered again on the second pass, and the specified dialect does not match the new current dialect, a warning is issued and compilation continues. If any errors were encountered on the first pass, the compiler will not attempt a second pass."

`#lang` may not be used in any compound statement, scope, or subroutine. However, it may be nested in module level preprocessor statements or used in an include file.

There is currently no restriction on where this directive may be placed in a source module. In future this may change, therefore best practice would be to use this directive before the first declaration, definition, or executable statement in the source.

This directive overrides the *-lang* option if it was given on the command line. However, if the *-forcelang* option was given on the command line, this directive will have no effect. A warning is issued,

the directive is ignored, and compilation will continue. This allows the user to explicitly override `#lang` directives.

Example

```
#lang "fbLite"
```

Differences from QB

- New to FreeBASIC

See also

- `$Lang`
- `__FB_LANG__`
- **Compiler Option: `-lang`**
- **Compiler Option: `-forcelang`**
- **FreeBASIC Dialects**

Preprocessor statement to add a search path for libraries

Syntax

```
#libpath "path"
```

Description

Adds a library search path to the linker's list of search paths as if it had been specified on the command line with the '-p' option.

Paths are relative to the working directory where fbc was invoked and relative to the directory of the source file.

No error is generated if the path does not exist and compilation and link will continue.

Example

```
' search the lib directory for external libraries  
#libpath "lib"
```

Differences from QB

- New to FreeBASIC

See also

- `#includ`
- `#include`
- **Compiler Option: -p**

Preprocessor directive to set the current line number and file name

Syntax

```
#line number [ "name" ]
```

Parameters

number
new line number
"name"
new file name (optional)

Description

Informs the compiler of a change in line number and file name and updates `__LINE__` macro values accordingly.

Both compile time messages and run-time messages are affected by

This directive allows other programs to generate source code for the program. It will have it return warning and/or error messages that refer to the original program.

Example

```
#line 155 "outside.src"

Error 1000

'' Output is:
'' Aborting due to runtime error 1000 at line 157
```

Differences from QB

- New to FreeBASIC

See also

- [__FILE__](#)
- [__LINE__](#)

#pragma



Preprocessor directive

Syntax

```
#pragma option [ = value ]  
Or  
#pragma push ( option [, value ] )  
Or  
#pragma pop ( option )
```

Parameters

Possible values for *option* and related *values*:

Option	Value	Description
msbitfields	0	Use bitfields compatible with gcc (default)
	-1 (or any other non-zero value)	Use bitfields compatible with those used in Microsoft
once	N/A	cause the source file in which the pragma appears to with #include once ...

If *value* is not given, the compiler assumes *-1 (TRUE)*.

Description

Allows the setting of compiler options inside the source code.

Push saves the current value of the *option* onto a stack, then assigns restores the *option* to its previous value, and removes it from the stack options to be changed for a certain part of source code, regardless of context, which is especially useful inside #include header files.

Example

```
' ' MSVC -
```

```
compatible bitfields: save the current setting and
#pragma push(msbitfields)

'' do something that requires MS-compatible bitfie

'' restore original setting
#pragma pop(msbitfields)
```

Differences from QB

- New to FreeBASIC

See also

- **#include**

Preprocessor diagnostic directive

Syntax

```
#print text
```

Description

Causes compiler to output *text* to screen during compilation.

Example

```
#print Now compiling module foo
```

Differences from QB

- New to FreeBASIC

See also

- [#error](#)

Preprocessor directive to undefine a macro

Syntax

```
#undef symbol
```

Description

Undefines a symbol previously defined with `#define`.

Can be used to ensure that a macro or symbol has a limited lifespan and does not conflict with a similar macro definition that may be defined later in the source code.

(Note: `#undef` should not be used to undefine variable or function names used in the current function scope. The names are needed internally by the compiler and removing them can cause strange and unexpected results.)

Example

```
#define ADD2(a_, b_) ((a_) + (b_))  
Print ADD2(1, 2)  
' Macro no longer needed so get rid of it ...  
#undef ADD2
```

Differences from QB

- New to Freebasic

See also

- `#define`
- `#macro`

- `#if`
- `#else`
- `#elseif`
- `#endif`
- `#ifdef`
- `#ifndef`
- `defined`

Metacommand to change the way arrays are allocated

Syntax

```
'$Dynamic  
or  
Rem $Dynamic
```

Description

'\$Dynamic is a metacommand that specifies that any following array declarations are variable-length, whether they are declared with constant subscript ranges or not. This remains in effect for the rest of the module in which **'\$Dynamic** is used, and can be overridden with **'\$Static**. It is equivalent to the **option Dynamic** statement.

Example

```
' compile with -lang fblite or qb  
  
#lang "fblite"  
  
'$DYNAMIC  
Dim a(100)  
' .....  
ReDim a(200)
```

Dialect Differences

- Only available in the *-lang fblite* and *-lang qb* dialects.

Differences from QB

- When used inside comments it must be the first token

See also

- `$Static`
- `Dim`
- `ReDim`
- `Erase`
- `Option Dynamic`

\$Include



Metacommand statement to include contents of another source file

Syntax

```
'$Include [once]: 'file'  
or  
Rem $Include [once]: 'file'
```

Description

\$Include inserts source code from another file at the point where the **\$Include** metacommand appears. This has the effect of compiling the source code from the include file as though it were part of the source file that includes it. Once the compiler has reached the end of the include file, the original source file continues to be compiled.

The **once** specifier tells the compiler to include the file only once even if it is included several times by the source code.

'**\$Include**': exists for compatibility with QuickBASIC. It is recommended to use **#include** instead.

Example

```
' header.bi file  
Type FooType  
    Bar As Byte  
    Barbeque As Byte  
End Type  
Dim Foo As FooType
```

```
' ' compile with -lang fblite or qb  
  
#lang "fblite"
```

```
' ' main.bas file

'$INCLUDE: "header.bi"

Foo.Bar = 1
Foo.Barbeque = 2
```

Dialect Differences

- Only available in the *-lang fblite* and *-lang qb* dialects.

Differences from QB

- None

See also

- `#include`

Metacommand to change the way arrays are allocated

Syntax

```
'$Static  
or  
Rem $Static
```

Description

'\$static is a metacommand that overrides the behavior of \$Dynamic, that is, arrays declared with constant subscript ranges are fixed-length. This remains in effect for the rest of the module in which '\$static is used, and can be overridden with \$Dynamic. It is equivalent to the `Option Static` statement.

Example

```
' compile with -lang fblite or qb  
  
#lang "fblite"  
  
'$dynamic  
Dim a(100)    '<<this array will be variable-  
length  
'$static  
Dim b(100)    '<<this array will be fixed-length
```

Dialect Differences

- Only available in the *-lang fblite* and *-lang qb* dialects.

Differences from QB

- When used inside comments it must be the first token

See also

- `$Dynamic`
- `Dim`
- `Erase`
- `ReDim`
- `Option Dynamic`
- `Option Static`

Metacommand statement to set the compiler dialect.

Syntax

```
'$lang: "lang"  
or  
Rem $lang: "lang"
```

Parameters

"lang"

The dialect to set, enclosed in double quotes, and must be one of "fb", "fb-lite", "qb", or "deprecated".

Description

If the *-forcelang* option was not given on the command line, **\$Lang** can be used to set the dialect for the source module in which it appears. At most two passes will be made on the source module. On the first pass: if the specified dialect is anything other than the default dialect (chosen with *-lang*, or "fb" by default), the compiler will reset the parser for another pass and restart compilation at the beginning of the source module. If this metacommand is encountered again on the second pass, and the specified dialect does not match the new current dialect, a warning is issued and compilation continues. If any errors were encountered on the first pass, the compiler will not attempt a second pass.

\$Lang may not be used in any compound statement, scope, or subroutine. However, it may be nested in module level preprocessor statements or used in an include file.

There is currently no restriction on where this directive may be placed in a source module. In future this may change, therefore best practice would be to use this directive before the first declaration, definition, or executable statement in the source.

This directive overrides the *-lang* option if it was given on the

command line. However, if the *-forcelang* option was given on the command line, this directive will have no effect. A warning is issued, the directive is ignored, and compilation will continue. This allows the user to explicitly override `$Lang` metacommands.

This metacommand was introduced in FreeBASIC version 0.20.0. Older versions of FB, and QuickBASIC, will treat it as a comment and silently ignore it.

Example

```
'$lang: "qb"
```

Differences from QB

- New to FreeBASIC
- QB handles '\$lang: as a normal comment

See also

- `#lang`
- `__FB_LANG__`
- **Compiler Option: -lang**
- **Compiler Option: -forcelang**
- **FreeBASIC Dialects**

Calculates the absolute value of a number

Syntax

```
Declare Function Abs ( ByVal number As Long ) As Long
Declare Function Abs ( ByVal number As ULong ) As ULong
Declare Function Abs ( ByVal number As LongInt ) As LongInt
Declare Function Abs ( ByVal number As ULongInt ) As ULongInt
Declare Function Abs ( ByVal number As Double ) As Double
```

Usage

```
result = Abs( number )
```

Parameters

number

Value to find the absolute value of.

Return Value

The absolute value of *number*.

Description

The absolute value of a number is its positive magnitude. If a number is negative, its value will be negated and the positive result returned. For example, `Abs(-1)` and `Abs(1)` both return 1. The required *number* argument can be any valid numeric expression.

Unsigned numbers will be treated as if they were signed, i.e. if the highest bit is set the number will be treated as negative, and its value negated.

The value returned will be greater than or equal to 0, with the exception of signed integers containing the lowest possible negative value that can be stored in its type, in which case negating it will overflow the result.

The `Abs` unary **operator** can be overloaded with user defined types.

Example

```
Dim n As Integer

Print Abs( -1 )
Print Abs( -3.1415 )
Print Abs( 42 )
Print Abs( n )

n = -69

Print Abs( n )
```

Output:

```
1
3.1415
42
0
69
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- None

See also

- [Sgn](#)
- [Operator](#)

Declare abstract methods

Syntax

```
Type typename Extends base_typename  
Declare Abstract Sub|Function|Property|Operator ...  
End Type
```

Description

Abstract is a special form of **Virtual**. The difference is that abstract methods do not have a body, but just the declaration. Essentially this allows the declaration of an interface which can be implemented by various derived types.

In order to call an abstract method, it must have been overridden and implemented by a derived data type, or else the program will abort. As a result, only types that implement all the abstract methods are allowed to create objects. For the same reason, a constructor should not call an unimplemented method.

Constructors cannot be abstract, since they cannot be virtual. In addition, abstract **Destructors** are not supported either, because a destructor body (no matter whether implicit or explicit) is needed in order to call base and field destructors.

Abstracts are called "pure virtual" in C++ (unlike FreeBASIC, C++ allows pure virtuals to have a body, but accessible only statically).

Note: In a multi-level inheritance, a same named method (same identifier and signature) can be declared **Abstract**, **Virtual** or normal (without specifier) at each inheritance hierarchy level. When there is mixing of specifiers, the usual order is abstract -> virtual -> normal, from top to bottom of the inheritance hierarchy. The access control (**Public/Protected/Private**) of an overriding method is not taken into account by the internal polymorphism process, but only for the initial call at compile-time.

A derived static method cannot override a base virtual/abstract method, but can shadow any base method (including virtual/abstract).

Example

```
Type Hello extends object
  Declare abstract Sub hi( )
End Type

Type HelloEnglish extends Hello
  Declare Sub hi( )
End Type

Type HelloFrench extends Hello
  Declare Sub hi( )
End Type

Type HelloGerman extends Hello
  Declare Sub hi( )
End Type

Sub HelloEnglish.hi( )
  Print "hello!"
End Sub

Sub HelloFrench.hi( )
  Print "Salut!"
End Sub

Sub HelloGerman.hi( )
  Print "Hallo!"
End Sub

Randomize( Timer( ) )

Dim As Hello Ptr h
```

```
For i As Integer = 0 To 9
  Select Case( Int( Rnd( ) * 3 ) + 1 )
  Case 1
    h = New HelloFrench
  Case 2
    h = New HelloGerman
  Case Else
    h = New HelloEnglish
  End Select

  h->hi( )
  Delete h
Next
```

Dialect Differences

- Only available in the *-lang fb* dialect.

Differences from QB

- New to FreeBASIC

See also

- **Virtual**
- **Type**
- **Extends**
- **Object**

Clause of the `open` statement to specify requested privileges

Syntax

```
Open filename for Binary Access {Read | Write | Read Write} as [
```

Usage

```
open filename for binary Access Read as #filenum
open filename for binary Access Write as #filenum
open filename for binary Access Read Write as #filenum
```

Parameters

Read
Open the file with only read privileges.

Write
Open the file with only write privileges.

Read Write
Open the file with read and write privileges.

Description

Access is used with the `open` statement to request read, write, or read & write access. If the Access clause is not specified, Read Write is assumed.

Example

This example shows how to open the file "data.raw" with `Read` and the `Binary` access, in `Binary` mode, in an open file number returned by `FreeFile`.

```
Dim As Integer o

' get an open file number.
o = FreeFile

' open file for read-only access.
Open "data.raw" For Binary Access Read As #o
```

```

'' make a buffer in memory thats the entire si
Dim As UByte file_char( LOF( o ) - 1 )

'' get the file into the buffer.
Get #o, , file_char()

Close

'' get another open file number.
o = FreeFile

'' open file for write-only access.
Open "data.out" For Binary Access Write As #o

'' put the buffer into the new file.
Put #o, , file_char()

Close

Print "Copied file ""data.raw"" to file ""data.c
Sleep

```

Differences from QB

- None known.

See also

- [Open](#)
- [Read](#)
- [Write](#)

Acos



Finds the arccosine of an angle

Syntax

```
Declare Function Acos ( ByVal number As Double ) As Double
```

Usage

```
result = Acos( number )
```

Parameters

number

A cosine value in the range [-1..1].

Return Value

The arccosine of *number*, in radians, in the range [0..Pi].

Description

Acos returns the arccosine of the argument *number* as a **Double** within the inverse of the **cos** function. The returned angle is measured in **rad**

Example

```
Dim h As Double
Dim a As Double
Input "Please enter the length of the hypotenuse c"
Input "Please enter the length of the adjacent side a"
Print ""
Print "The angle between the sides is"; Acos ( a / h ) * 180 / Pi
Sleep
```

The output would look like:

```
Please enter the length of the hypotenuse of a triangle: 5
```

Please enter the length of the adjacent side of the triangle:

The angle between the sides is 0.6435011087932843

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- [Cos](#)
- [A Brief Introduction To Trigonometry](#)

Add



Parameter to the **Put** graphics statement which selects addition as the b

Syntax

```
Put [ target, ] [ STEP ] ( x,y ), source [ ,( x1,y1 )-( x2,y2 )  
,multiplier ]
```

Parameters

Add

Required.

multiplier

Optional value between 0 and 255. The source pixels are premultiplied ($multiplier / 256$) before being added. If omitted, this value defaults

Description

Add selects addition as the method for blitting an image buffer. For each target pixel, the values of each respective component are added together to produce the result.

The addition is saturated - i.e. if the sum of the two values is 256 or more, the result will be cropped down to 255.

This method will work in all color modes. Mask colors (color 0 for indexed images) will be skipped, though values of 0 (**RGBA**(0, 0, 0, 0)) will have no effect.

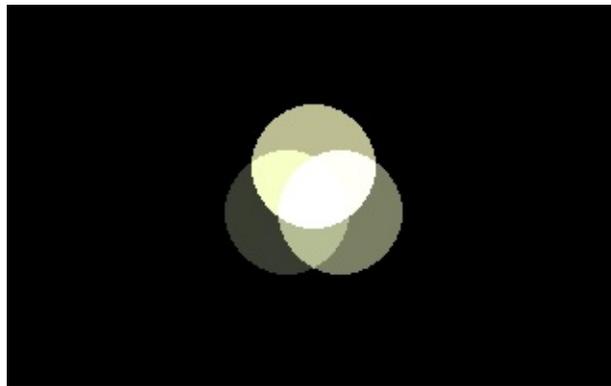
Example

```
'open a graphics window  
ScreenRes 320, 200, 16  
  
'create a sprite containing a circle  
Const As Integer r = 32  
Dim c As Any Ptr = ImageCreate(r * 2 + 1, r * 2 + 1, Circle c, (r, r), r, RGB(255, 255, 192), , , 1, f
```

```
'put the sprite at three different multiplier
''levels, overlapping each other in the middle
Put (146 - r, 108 - r), c, add, 64
Put (174 - r, 108 - r), c, add, 128
Put (160 - r, 84 - r), c, add, 192

''free the memory used by the sprite
ImageDestroy c

''pause the program before closing
Sleep
```



Differences from QB

- New to FreeBASIC

See also

- [Trans](#)
- [Alpha](#)
- [Custom](#)
- [Put \(Graphics\)](#)

Alias



Clause of the **Sub** and **Function** statements that provides an alternate int

Syntax

```
[Declare] { Sub | Function } usablename Alias "alternatename" (.
```

Usage

```
declare sub usablename Alias "alternatename" ( ... )  
or  
declare function usablename Alias "alternatename" ( ... )  
or  
sub usablename Alias "alternatename" ( ... )  
...  
end sub  
or  
function usablename Alias "alternatename" ( ... )  
...  
end function
```

Description

`Alias` gives an alternate name to a procedure. This alternate name can (if the function is not private) to the linker when linking with code written

`Alias` is commonly used for procedures in libraries written in other languages but invalid in BASIC. When using `Alias` with `Declare`, only the alternate

Differently from normal procedure names, `Alias` does not change the case of an exported function with a particular name or with a particular case.

Example

If there is a sub called `xClearScreen` in an external library and you want to do so:

```
Declare Sub ClearVideoScreen Alias "xClearScreen" (
```

A procedure meant to be used by external C code, exported as MyExport

```
Function MultiplyByFive cdecl Alias "MyExportedProc"  
    Return Parameter * 5  
End Function
```

Differences from QB

- In QB, Alias only worked with **Declare**.

See also

- **Declare**
- **Export**

Allocate



Allocates a block of memory from the free store

Syntax

```
Declare Function Allocate cdecl ( ByVal count As UInteger ) As A
```

Usage

```
result = Allocate( count )
```

Parameters

count

The size, in bytes, of the block of memory to allocate.

Return Value

If successful, the address of the start of the allocated memory is returned. If *count* < 0, then the null pointer (0) is returned.

Description

Attempts to allocate, or reserve, *count* number of bytes from the free store.

As the initial value of newly allocated memory is unspecified, **Allocate** returns a string, because the string descriptor being not cleared (containing random data). To avoid writing to a random place in memory or trying to deallocate a random string, use **CAllocate** (clearing memory), or **New** (calling **Clear** to explicitly clear the descriptor (setting to 0) before the first string use.

The pointer that is returned is an **Any Ptr** and points to the start of the allocated memory, even if *count* is zero.

Allocated memory must be deallocated, or returned back to the free store.

Example

```

'' This program uses the ALLOCATE(...) function to
'' then filled with the first 15 numbers of the Fi
'' screen. Note the call to DEALLOCATE(...) at the

Const integerCount As Integer = 15

'' Try allocating memory for a number of integ
''
Dim buffer As Integer Ptr
buffer = Allocate(integerCount * SizeOf(Integer))

If (0 = buffer) Then
    Print "Error: unable to allocate memory, c
    End -1
End If

'' Prime and fill the memory with the fibonacc
''
buffer[0] = 0
buffer[1] = 1
For i As Integer = 2 To integerCount - 1
    buffer[i] = buffer[i - 1] + buffer[i - 2]
Next

'' Display the sequence.
''
For i As Integer = 0 To integerCount - 1
    Print buffer[i] ;
Next

Deallocate(buffer)
End 0

```

Output is:

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

It is important to free allocated memory if it's not going to be used any more, and if the address of that memory is somehow overwritten or forgotten, it is known as a memory leak, and should be avoided at all costs. Note that when an application terminates, either by an "ordinary" exit or crash, so the leak is neverthere nevertheless it's a good habit to free any allocated memory inside your functions. In a function with a memory leak, where the address of allocated memory is not saved, and the function is called frequently, the total amount of memory wasted can be significant.

```
' ' Bad example of Allocate usage, causing memory leak
Sub BadAllocateExample()
    Dim p As Byte Ptr
    p = Allocate(420) ' assign pointer to new memory
    p = Allocate(420) ' reassign same pointer to new memory
                        ' old address is lost and memory is leaked
    Deallocate(p)
End Sub

' ' Main
BadAllocateExample() ' Creates a memory leak
Print "Memory leak!"
BadAllocateExample() ' ... and another
Print "Memory leak!"
End
```

Platform Differences

- This procedure is not guaranteed to be thread-safe.

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- **CAllocate**
- **Reallocate**
- **Deallocate**

Alpha



Parameter to the **Put** graphics statement which selects alpha blending a

Syntax

```
Put [ target, ] [ STEP ] ( x,y ), source [ ,( x1,y1 )-( x2,y2 )  
Put [ target, ] [ STEP ] ( x,y ), source [ ,( x1,y1 )-( x2,y2 )
```

Parameters

Alpha

Required.

alphaval

Optional alpha parameter in the range [0..255]. Overrides alpha value

Description

Alpha selects alpha blending as the method for **Put**ting an image. If the mask color (magenta) will be treated as transparent

If *alphaval* is not specified, **Alpha** will only work in 32-bit color depth, and pixels using the mask color will be treated as normal, and drawn with

Alpha also has another mode which allows an 8-bit image to be **Put** into a channel of the 32-bit image with the contents of the 8-bit image.

Alpha values range between 0 and 255. An alpha value of 0 will not do anything, and an alpha value of 255 will get a range between 2 and 256, and the result is then divided by 255 to get the exact value of each pixel from the source and destination pixels. In alpha blending mode, 0 is equivalent to doing nothing at all, and all the other alpha values

Example

This example compares the two different **Alpha** modes, including how

```
' ' Set up a 32-bit screen  
ScreenRes 320, 200, 32  
  
' ' Draw checkered background
```

```

For y As Integer = 0 To 199
    For x As Integer = 0 To 319
        PSet (x, y), IIf((x Shr 2 Xor y Shr 2) And
    Next x
Next y

'' Make image sprite for Putting
Dim img As Any Ptr = ImageCreate(32, 32, RGBA(0, 0, 0, 0))
For y As Single = -15.5 To 15.5
    For x As Single = -15.5 To 15.5
        Dim As Integer r, g, b, a
        If y <= 0 Then
            If x <= 0 Then
                r = 255: g = 0: b = 0 '' red
            Else
                r = 0: g = 0: b = 255 '' blue
            End If
        Else
            If x <= 0 Then
                r = 0: g = 255: b = 0 '' green
            Else
                r = 255: g = 0: b = 255 '' magenta
            End If
        End If
        a = 255 - (x ^ 2 + y ^ 2)
        If a < 0 Then a = 0: r = 255: g = 0: b = 0
        PSet img, (15.5 + x, 15.5 - y), RGBA(r, g, b, a)
    Next x
Next y

'' Put with single Alpha value, Trans for comparison
Draw String (32, 10), "Single alpha"
Put (80 - 16, 50 - 16), img, Alpha, 64
Put (80 - 16, 100 - 16), img, Alpha, 192
Put (80 - 16, 150 - 16), img, Trans

'' Put with full Alpha channel
Draw String (200, 10), "Full alpha"
Put (240 - 16, 100 - 16), img, Alpha

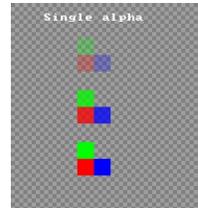
```

```

'' Free the image memory
ImageDestroy img

'' Wait for a keypress
Sleep

```



This example shows the special method for setting a 32-bit alpha cha

```

Dim As Any Ptr img8, img32
Dim As Integer x, y, i

'' Set up an 8-bit graphics screen
ScreenRes 320, 200, 8
For i = 0 To 255
    Palette i, i, i, i
Next i
Color 255, 0

'' Create an 8-bit image
img8 = ImageCreate(64, 64, 0, 8)
For y = 0 To 63
    For x = 0 To 63
        Dim As Single x2 = x - 31.5, y2 = y - 31.5
        Dim As Single t = Sqr(x2 ^ 2 + y2 ^ 2) / 5
        PSet img8, (x, y), Sin(t) ^ 2 * 255
    Next x
Next y

Draw String (16, 4), "8-bit Alpha sprite"
Put (16, 16), img8
Sleep

```

```

'' Set up a 32-bit graphics screen
ScreenRes 320, 200, 32
For y = 0 To 199
    For x = 0 To 319
        PSet (x, y), IIf(x - y And 3, RGB(160, 160, 160))
    Next x
Next y

'' Create a 32-bit, fully opaque sprite
img32 = ImageCreate(64, 64, 0, 32)
For y = 0 To 63
    For x = 0 To 63
        PSet img32, (x, y), RGB(x * 4, y * 4, 128)
    Next x
Next y

Draw String (16, 4), "Original Alpha channel"
Put (16, 16), img32, Alpha

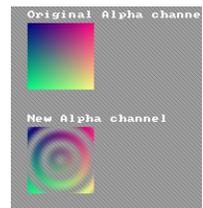
'' Put a new alpha channel using the 8-bit image
Put img32, (0, 0), img8, Alpha

Draw String (16, 104), "New Alpha channel"
Put (16, 116), img32, Alpha

''Free the memory for the two images
ImageDestroy img8
ImageDestroy img32

Sleep

```



Differences from QB

- New to FreeBASIC

See also

- [Put \(Graphics\)](#)
- [Trans](#)
- [Custom](#)

Operator And (Conjunction)



Returns the bitwise-and (conjunction) of two numeric values

Syntax

```
Declare Operator And ( ByRef lhs As T1, ByRef rhs As T2 ) As Ret
```

Usage

```
result = lhs And rhs
```

Parameters

lhs

The left-hand side expression.

T1

Any numeric or boolean type.

rhs

The right-hand side expression.

T2

Any numeric or boolean type.

Ret

A numeric or boolean type (varies with *T1* and *T2*).

Return Value

Returns the bitwise-and (conjunction) of the two operands.

Description

This operator returns the bitwise-and of its operands, a logical operation on two operands (for conversion of a boolean to an integer, false or true boolean).

The truth table below demonstrates all combinations of a boolean-and

Lhs Bit	Rhs Bit	Result
0	0	0
1	0	0
0	1	0
1	1	1

1	1	1
---	---	---

No short-circuiting is performed - both expressions are always evaluated.

The return type depends on the types of values passed. `Byte`, `UByte` and right-hand side types differ only in signedness, then the return type of the two types is returned. Only if the left and right-hand side types are the same, the left-hand side type is returned.

This operator can be overloaded for user-defined types.

Example

```
' Using the AND operator on two numeric values
Dim As UByte numeric_value1, numeric_value2
numeric_value1 = 15 '00001111
numeric_value2 = 30 '00011110

'Result = 14 = 00001110
Print numeric_value1 And numeric_value2
Sleep
```

```
' Using the AND operator on two conditional expressions
Dim As UByte numeric_value1, numeric_value2
numeric_value1 = 15
numeric_value2 = 25

If numeric_value1 > 10 And numeric_value1 < 20 Then
  Print "Numeric_Value1 is between 10 and 20"
If numeric_value2 > 10 And numeric_value2 < 20 Then
  Print "Numeric_Value2 is between 10 and 20"
Sleep
```

```
' This will output "Numeric_Value1 is between 10 and 20"
' both conditions of the IF statement is true
' It will not output the result of the second IF statement
' because the first condition is true and the second is false.
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- None

See also

- [AndAlso](#)
- [Operator Truth Tables](#)

Operator AndAlso (Short Circuit Conjunction)



Returns the short circuit-and (conjunction) of two numeric values

Syntax

```
Declare Operator AndAlso ( ByRef lhs As T1, ByRef rhs As T2 ) As
```

Usage

```
result = lhs AndAlso rhs
```

Parameters

lhs

The left-hand side expression.

T1

Any numeric or boolean type.

rhs

The right-hand side expression.

T2

Any numeric or boolean type.

Ret

A numeric or boolean type (varies with *T1* and *T2*).

Return Value

Returns the short circuit-and (conjunction) of the two operands.

Description

This operator evaluates the left hand side expression. If the result is z immediately returned. If the result is nonzero then the right hand side the logical result from that is returned.

(for conversion of a boolean to an integer, false or true boolean value integer value)

The truth table below demonstrates all combinations of a short circuit- ' denotes that the operand is not evaluated.

--	--	--

Lhs Value	Rhs Value	Result
0	-	0
nonzero	0	0
nonzero	nonzero	-1

Short-circuiting is performed - only expressions needed to calculate the result are evaluated.

The return type is almost always an **Integer**, of the value 0 or -1, denoting false or true respectively. Except if the left and right-hand side types are both **Boolean**, the return type is also **Boolean**.

This operator cannot be overloaded for user-defined types.

Example

```
' ' Using the ANDALSO operator to guard against array access
' ' when the index is out of range

Dim As Integer isprime(1 To 10) = { _
    _ ' 1 2 3 4 5 6 7 8 9 10
      0, 1, 1, 0, 1, 0, 1, 0, 0, 0 _
    }

Dim As Integer n
Input "Enter a number between 1 and 10: ", n

' isprime() array will only be accessed if n is in range
If (n >= 1 And n <= 10) AndAlso isprime(n) Then
    Print "n is prime"
Else
    Print "n is not prime, or out of range"
End If
```

Differences from QB

- This operator was not available in QB.

See also

- [OrElse](#)
- [And](#)
- [Operator Truth Tables](#)

And



Parameter to the **Put** graphics statement which uses a bit-wise **And** as the

Syntax

```
Put [ target, ] [ STEP ] ( x,y ), source [ ,( x1,y1 )-( x2,y2 )
```

Parameters

And
Required.

Description

The **And** method combines each source pixel with the corresponding color pixel using a bit-wise **And** function. The result of this is output as the destination pixel. This method works in all graphics modes. There is no mask color, although all bits set (255 for 8-bit palette modes, or **RGBA**(255, 255, 255, 255) in full-color modes) have no effect, because of the behavior of **And**.

In full-color modes, each component (red, green, blue and alpha) is kept separate, so the operation can be made to only affect some of the channels, by setting values of the other channels to 255.

Example

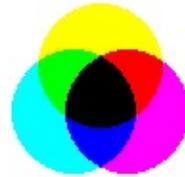
```
'open a graphics window
ScreenRes 320, 200, 16
Line (0, 0)-(319, 199), RGB(255, 255, 255), bf

'create 3 sprites containing cyan, magenta and yellow
Const As Integer r = 32
Dim As Any Ptr cc, cm, cy
cc = ImageCreate(r * 2 + 1, r * 2 + 1, RGBA(255, 255, 0, 255))
cm = ImageCreate(r * 2 + 1, r * 2 + 1, RGBA(255, 0, 255, 255))
cy = ImageCreate(r * 2 + 1, r * 2 + 1, RGBA(255, 255, 255, 255))
Circle cc, (r, r), r, RGB(0, 255, 255), , , 1, f
Circle cm, (r, r), r, RGB(255, 0, 255), , , 1, f
```

```
Circle cy, (r, r), r, RGB(255, 255, 0), , , 1, f
''put the three sprites, overlapping each other in
Put (146 - r, 108 - r), cc, And
Put (174 - r, 108 - r), cm, And
Put (160 - r, 84 - r), cy, And

''free the memory used by the sprites
ImageDestroy cc
ImageDestroy cm
ImageDestroy cy

''pause the program before closing
Sleep
```



Differences from QB

- None

See also

- [And](#)
- [Put \(Graphics\)](#)

Any is used as a placeholder for a type or value in various ways.

Syntax

```

Dim identifier As Any Pointer|Ptr
Or
Declare Sub|Function identifier ( ByRef identifier As Any [ , ..
Or
Dim identifier(Any [, Any...]) As DataType
Or
[ Declare ] { Sub | Function } proc_name ( param(Any [, Any...])
Or
Dim identifier As DataType = Any
Or
New DataType ( Any )
Or
New(Address) DataType [count] { Any }
Or
InStr|InStrRev ( string, Any substring )

```

Description

- Pointers:

A special pointer type called the **Any Ptr** (or "Any **Pointer**") allows pointer arithmetic as an instance of *DataType*. Pointer arithmetic is allowed on an **Any Ptr**.

A pure **Any Ptr** has no type checking by the compiler. It can be implicitly converted to any other data type.

Any on its own is not a valid data type for a variable. Also, it is illegal to use **Any** as a data type in a procedure declaration.

This should not be confused with **variant**, a Visual Basic data type which can hold any type of data.

- Byref parameters:

Any can be used in procedure prototypes (in a **Declare** statement) with **ByRef** parameters. It is deprecated and it only exists for compatibility with QB.

- Array dimensions:

In array declarations, **Any** can be specified in place of the array bound. If **Any** is used, the array bound of **Any**s specified (use the syntax with **Any** is mandatory when declaring an array).

In parameter declarations, `Any` can be also specified instead of empty

- Initialization:

`Any` can be used as a fake initializer to disable the default initialization program's responsibility to fill the variables with meaningful data before

Comparison to C/C++: This matches the behavior of a variable declar

Similar to `Any` initializers for variables, `Any` can also be used with the `Nil` (that do not have constructors).

- Instr/InstrRev:

`Any` can be used with `InStr` or `InStrRev` as a qualifier for the *substring*

Example

```
Declare Sub echo(ByVal x As Any Ptr) '' echo will

Dim As Integer a(0 To 9) = Any '' this variable is
Dim As Double d(0 To 4)

Dim p As Any Ptr

Dim pa As Integer Ptr = @a(0)
Print "Not initialized ";
echo pa '' pass to echo a pointer to integer

Dim pd As Double Ptr = @d(0)
Print "Initialized ";
echo pd '' pass to echo a pointer to double

p = pa '' assign to p a pointer to integer
p = pd '' assign to p a pointer to double

Sleep

Sub echo (ByVal x As Any Ptr)
    Dim As Integer i
    For i = 0 To 39
```

```

        'echo interprets the data in the pointer a
        Print Cast(UByte Ptr, x)[i] & " ";
    Next
    Print
End Sub

```

```

'Example of ANY disabling the variable type checking
Declare Sub echo (ByRef a As Any) ' ANY disables

Dim x As Single
x = -15
echo x ' Passing a single to a function
Sleep

Sub echo (ByRef a As Integer)
    Print Hex(a)
End Sub

```

```

Dim a(Any) As Integer ' 1-dimensional dynamic array
Dim b(Any, Any) As Integer ' 2-dimensional dynamic array
Dim c(Any, Any, Any) As Integer ' 3-dimensional dynamic array
' etc.

' Further Redims or array accesses must have a matching number of dimensions
ReDim a(0 To 1) As Integer
ReDim b(1 To 10, 2 To 5) As Integer
ReDim c(0 To 9, 0 To 5, 0 To 1) As Integer

```

Dialect Differences

- Not available in the *-lang qb* dialect.

Differences from QB

- Pointers and initializers are new to FreeBASIC.

See also

- [Dim](#)
- [Declare](#)

Specifies text file to be opened for append mode

Syntax

```
Open filename for Append [Encoding encoding_type] [Lock  
lock_type] as [#]filenum
```

Parameters

filename

file name to open for append

encoding_type

indicates encoding type for the file

lock_type

locking to be used while the file is open

filenum

unused file number to associate with the open file

Description

A file mode used with **open** to open a text file for writing.

This mode is used to add text to an existing file with **Print #**, or comma separated values with **write#**.

Text files can't be simultaneously read and written in FreeBASIC, so if both functions are required on the same file, it must be opened twice.

filename must be a string expression resulting in a legal file name in the target OS, without wildcards. The file will be sought for in the present directory, unless the *filename* contains a path . If the file does not exist, it is created. The pointer is set after the last character of the file.

Encoding_type indicates the Unicode **Encoding** of the file, so characters are correctly read. If omitted, "ascii" encoding is defaulted. Only little endian character encodings are supported at the moment.

- "utf8"

- "utf16"
- "utf32"
- "ascii" (the default)

Lock_type indicates the way the file is locked for other processes, it is one of:

- **Read** - the file can be opened simultaneously by other processes, but not for reading
- **write** - the file can be opened simultaneously by other processes, but not for writing
- **Read write** - the file cannot be opened simultaneously by other processes (the default)

filenum is a valid FreeBASIC file number (in the range 1..255) not being used for any other file presently open. The file number identifies the file for the rest of file operations. A free file number can be found using the **FreeFile** function.

Example

```
Dim i As Integer
For i = 1 To 10
  Open "test.txt" For Append As #1
  Print #1, "extending test.txt"
  Close #1
Next
```

Differences from QB

- None

See also

- **Input (File Mode)**
- **Open**

- **Output**
- **(Print | ?) #**
- **Write #**

Optional part of a declaration which specifies a data type, or part of the `Open` statement which specifies a file handle.

Syntax

```
symbolName As datatype
```

```
Open ... As #filenumber
```

```
Type ... As datatype
```

Description

`As` is used to declare the type of variables, fields or arguments and is used in the `Open` statement to determine the file handle. `As` is also used with the `Type` statement to emulate C's typedef statement.

Example

```
' don't try to compile this code, the examples are for illustration only
Declare Sub mySub (X As Integer, Y As Single, Z As String)
' ...

Dim X As Integer
' ...

Type myType
    X As Integer
    Y As Single
    Z As String
End Type
' ...

Type TheNewType As myType
' ...

Open "test" For Input As #1
' ...
```

Differences from QB

- The **Type (Alias)** syntax was not supported in QB.

See also

- **Declare**
- **Dim**
- **Type**
- **Open**

Assert



Debugging macro that halts program execution if an expression is evalu

Syntax

```
#define Assert(expression) If (expression) = 0 Then : fb_Assert(  
#expression ) : End If
```

Usage

```
Assert( expression )
```

Parameters

expression

Any valid conditional/numeric expression. If *expression* evaluates to 0

Description

The **Assert** macro is intended for use in debugging and works only if t command line. In this case it prints an error message and stops the p evaluates to 0.

Its normal use is to check the correct value of the variables during del

If -g is not passed to fbc, the macro does not generate any code, and

Note: If an **Assert** fails while the program is in a graphics **screen**, the e be printed to the graphics screen, which will be closed immediately af

Example

```
Sub foo  
  Dim a As Integer  
  a=0  
  Assert(a=1)  
End Sub
```

```
foo
```

```
'' If -
```

```
g is used this code stops with: test.bas(3): asser
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- [#Assert](#)
- [AssertWarn](#)

AssertWarn



Debugging macro that prints a warning if an expression evaluates to 0.

Syntax

```
#define AssertWarn(expression) If (expression) = 0 Then : fb_Ass  
__FUNCTION__, #expression ) : End If
```

Usage

```
AssertWarn( expression )
```

Parameters

expression

Any valid expression. If *expression* evaluates to 0, a warning message

Description

The `AssertWarn` macro is intended for use in debugging and works on the FBC command line. In this case it prints a warning message if *expression* evaluates to 0, the program execution like `Assert` does.

Its normal use is to check the correct value of the variables during development.

If `-g` is not passed to `fbcc`, the macro does not generate any code.

Example

```
Sub foo  
    Dim a As Integer  
    a=0  
    AssertWarn(a=1)  
End Sub  
  
foo  
  
' ' If -
```

```
g is used this code prints: test.bas(3): assertior
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- [Assert](#)

Returns the corresponding ASCII or Unicode integer representation of a character

Syntax

```
Declare Function Asc ( ByRef str As Const String, ByVal position
Integer = 1 ) As ULong
Declare Function Asc ( ByVal str As Const ZString Ptr, ByVal
position As Integer = 1 ) As ULong
Declare Function Asc ( ByVal str As Const WString Ptr, ByVal
position As Integer = 1 ) As ULong
```

Usage

```
result = Asc( str [, position ] )
```

Parameters

str

The source string.

position

The position in the string of a character.

Return Value

The raw character value stored at *position* in *str*.

Description

If *str* is a **String** or a **ZString**, the **UByte** value at that *position* is returned. This will be a 7-bit **ASCII** code, or even a 8-bit character value from some code-page, depending on the string data stored in *str*.

If *str* is a **WString**, the **UShort** (Windows) or **ULong** (Linux) value at that *position* is returned. This will be a 16bit value on Windows (WStrings use UTF16 there), or a 32bit value on Linux (WStrings use UTF32 there).

The function returns zero (0) if the string is a zero length string, *position* is less than one (1), or *position* is greater than the number of characters.

str.

`chr` performs the opposite function for ASCII strings, while `wchr` is the opposite for Unicode strings, returning a string containing the character represented by the code passed as an argument.

Example

```
Print "the ascii code of 'a' is: "; Asc("a")
Print "the ascii code of 'b' is: "; Asc("abc", 2)
```

will produce the output:

```
the ascii code of 'a' is: 97
the ascii code of 'b' is: 98
```

Unicode example (Note to documentation editors: don't put inside `%%` (qbasic) markers or the Russian text will disappear!)

will produce the output:

```
dim a as wstring * 11
a = "Привет, мир"
print "the Unicode of the second char of " & a & " is: " & asc(
```

```
the Unicode of the second char of Привет, мир is: 1088
```

Platform Differences

- DOS does not support the wide-character string version of `Asc`.

Differences from QB

- The optional *position* argument is new to FreeBASIC.

- QB does not support the wide-character string version of `Asc`

See also

- **ASCII Character Codes**
- `Chr`
- `Str`
- `Val`

Asin



Finds the arcsine of a number

Syntax

```
Declare Function Asin ( ByVal number As Double ) As Double
```

Usage

```
result = Asin( number )
```

Parameters

number

Sine value in the range [-1..1].

Return Value

The arcsine of *number*, in radians, in the range [-Pi/2..Pi/2].

Description

Asin returns the arcsine of the argument *number* as a **Double** within the the inverse of the **sin** function. The returned angle is measured in **rad**

Example

```
Dim h As Double
Dim o As Double
Input "Please enter the length of the hypotenuse c
Input "Please enter the length of the opposite side o
Print ""
Print "The angle between the sides is"; Asin ( o /
Sleep
```

The output would look like:

Please enter the length of the hypotenuse of a triangle: 5
Please enter the length of the opposite side of the triangle:
The angle between the sides is 0.6435011087932844

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- [Sin](#)
- [A Brief Introduction To Trigonometry](#)

Code block that allows the use of architecture-specific instructions.

Syntax

```
Asm  
  architecture-dependent instructions  
End Asm
```

Or

Asm *architecture-dependent instructions*

Description

The **Asm** block is used to insert specific machine-code instructions in a program operations that cannot be carried out using the features of the language performance-sensitive sections of code.

The current FreeBASIC compiler currently only produces code for Intel architecture, however, in the future, the compiler might be ported to a platform which uses a different instruction set. Therefore, **Asm** blocks should only be used when necessary and an alternative should be provided if possible.

The return value of a function may be set by using the **Function** keyword, see the example below.

Asm block comments have the same syntax as usual FreeBASIC **Comment** comments, not " ; " as usual in assembly code.

x86 Specific:

Syntax

The syntax of the inline assembler is a simplified form of Intel syntax. In the majority of x86 assemblers, such as MASM, TASM, NASM, YASM and GAS, the destination of an instruction is placed first, followed by the source. A variable or label may be referenced in an **Asm** block. The assembler used by FreeBASIC supports the `.intel_syntax noprefix` directive, and **Asm** blocks are passed through unmodified, with substitution of local variable names for stack frame references, and con-

Instruction syntax is mostly the same as FASM uses, one important difference is that the word "ptr" is needed by GAS to indicate pointer settings to be followed by the word "ptr".

```
' Assuming "n" is a FB global or local ULONG variable
mov  eax, [n]          ' OK: size is apparent from operand
inc  [n]              ' Not OK: size is not given
inc  dword [n]        ' Not OK: size given, but still
inc  dword Ptr [n]    ' OK: "ptr" is needed by GAS to
```

Register Preservation

When an `asm` block is opened, the registers `ebx`, `esi`, and `edi` are pushed onto the stack. When the block is closed, these registers are popped back from the stack. This is because they are preserved by most or all OS's using the x86 CPU. You can therefore explicitly preserve them yourself. You should not change `esp` and `ebp`, which address local variables.

Register Names

The names of the registers for the x86 architecture are written as follows:

- 4-byte integer registers: `eax`, `ebx`, `ecx`, `edx`, `ebp`, `esp`
- 2-byte integer registers: `ax`, `bx`, `cx`, `dx`, `bp`, `sp`, `di`, `si` (registers)
- 1-byte integer registers: `al`, `ah`, `bl`, `bh`, `cl`, `ch`, `dl`, `dh` (registers)
- Floating-point registers: `st(0)`, `st(1)`, `st(2)`, `st(3)`,
- MMX registers (aliased onto floating-point registers: `mm6`, `mm7`)
- SSE registers: `xmm0`, `xmm1`, `xmm2`, `xmm3`, `xmm4`, `xmm5`, `xmm6`, `xmm7`

Instruction Set

See these external references:

- [Original Intel 80386 manual from 1986](#)
- [Latest Intel Pentium 4 manuals](#)
- [NASM x86 Instruction Reference](#) (Please note that this is for NASM, not GAS)

used by FreeBASIC, but this page provides a good

Unsafe instructions

Note that the FreeBASIC compiler produces 32-bit protected-mode code in an unprivileged user level; therefore, privileged and sensitive instructions possibly won't work correctly or cause a runtime "General Protection Fault" (SIGILL error). The following are the privileged and sensitive instructions on Xeon:

- cli *1
- clts
- hlt
- in *1
- ins *1
- int *1
- into *1
- invd
- invlpg
- lgdt
- lidt
- lldt
- lmsw
- ltr
- mov to/from CRn, DRn, TRn
- out *1
- outs *1
- rdmsr
- rdpmc *2
- rdtsc *2
- sti *1
- str
- wbinvd
- wrmsr
- all SSE2 and higher instructions *2

*1: sensitive to IOPL, fine in DOS

*2: sensitive to permission bits in CR4, see below

The privileged instructions will work "correctly" in DOS when running on (non-default) Ring 0 version of CWSDPMI, WDOSX or D3X, nevertheless useful and dangerous when executed from DPMI code. RDTSC (Read Time Stamp Counter) is shown to be allowed by most, or all OS'es.

However the usefulness of RDTSC has been diminished with the advent of SSE2 and higher instructions are disabled "by default" after CPL. Linux usually do enable them, in DOS it is business of the DPMI host: CWSDPMI won't. The INT instruction is usable in the DOS version/target differently from real mode DOS, see also FaqDOS.

The segment registers (cs, ds, es, fs, gs) should not be changed from real mode cases with the DOS port (note that they do NOT work the same way as FaqDOS). The operating system or DPMI host is responsible for memory segments (selectors) in protected mode is very different from real-mode

Note that those "unsafe" instructions are not guaranteed to raise a "privilege insufficient" exception - the OS or DPMI host can decide to "emulate" them from some CRx works under HDPMI32), or "dummy" (nothing happens, a NOP).

Example

```
' ' This is an example for the x86 architecture.
Function AddFive(ByVal num As Long) As Long
    Asm
        mov eax, [num]
        add eax, 5
        mov [Function], eax
    End Asm
End Function

Dim i As Long = 4

Print "4 + 5 ="; AddFive(i)
```

4 + 5 = 9

FreeBASIC's Assembler is AS / GAS, the assembler of GCC, so an extra apply:

- The error lines returned by FBC for `Asm` blocks are not really simply displays the errors returned by AS , the lines are re make FreeBASIC preserve them, the compiler must be in delete ASM files").
- The label names are case sensitive inside `Asm` blocks.

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the `z`

Differences from QB

- New to FreeBASIC

See also

- [Function](#)
- [Naked](#)

Atan2



Returns the arctangent of a ratio

Syntax

```
Declare Function Atan2 ( ByVal y As Double, ByVal x As Double )
```

Usage

```
result = ATan2( y, x )
```

Parameters

y
Vertical component of the ratio.
x
Horizontal component of the ratio.

Return Value

The angle whose tangent is y/x , in radians, in the range $[-\text{Pi}..\text{Pi}]$.

Description

ATan2 returns the arctangent of the ratio y/x as a **Double** within the range $[-\text{Pi}..\text{Pi}]$. The returned angle is measured in **degrees**).

Example

```
Print Atan2 ( 4, 5 )      'this is the same as PRINT
```

The output would be:

```
0.6747409422235527
```

Differences from QB

- New to FreeBASIC

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

See also

- [Tan](#)
- [Atn](#)
- [A Brief Introduction To Trigonometry](#)

Atn



Returns the arctangent of a number

Syntax

```
Declare Function Atn ( ByVal number As Double ) As Double
```

Usage

```
result = Atn( number )
```

Parameters

number
A number.

Return Value

The angle, in radians, whose tangent is *number*, in the range [-Pi/2..Pi/2].

Description

Atn returns the arctangent of the argument *number* as a **Double** within the range of **-Pi/2** to **Pi/2**. The arctangent is the inverse of the **Tan** function. The returned angle is measured in **radians** (not **degrees**).

Example

```
Print "Pi ="; Atn ( 1.0 ) * 4  
Print Atn ( 4 / 5 )
```

The output would be:

```
Pi = 3.141592653589793  
0.6747409422235527
```

Differences from QB

- None

See also

- [Tan](#)
- [Atan2](#)
- [A Brief Introduction To Trigonometry](#)

Base (Initializer)



Specifies an initializer for the base UDT in derived **Udt** constructors

Syntax

```
Base ( constructor-parameters... )  
or:  
Base UDT-initializer
```

Description

The **Base** initializer can be used at the top of constructors of derived U allows to specify an explicit constructor call or UDT initializer to be use initialize the base object. It will replace the implicit default initialization must appear above any other statements in the constructor it is used

Note: Unlike "**Base**()", a "**Base.Constructor**()" statement does not re the implicit default initialization done by the constructor of a derived U can usually not be used legally, because it would result in two constru for the base object.

Example

```
Type SimpleParent  
    As Integer a, b, c  
End Type  
  
Type Child extends SimpleParent  
    Declare Constructor( )  
End Type  
  
Constructor Child( )  
    ' Simple UDT initializer  
    Base( 1, 2, 3 )  
End Constructor
```

```

Type ComplexParent
  As Integer i
  Declare Constructor( ByVal As Integer = 0 )
End Type

Constructor ComplexParent( ByVal i As Integer = 0
  this.i = i
End Constructor

Type Child extends ComplexParent
  Declare Constructor( )
  Declare Constructor( ByRef As Child )
End Type

Constructor Child( )
  ' Base UDT constructor call
  Base( 1 )
End Constructor

Constructor Child( ByRef rhs As Child )
  ' Base UDT constructor call
  Base( rhs.i )
End Constructor

```

Dialect Differences

- Methods are only supported in the *-lang fb* dialect, hence `Base` function in other dialects.

Differences from QB

- New to FreeBASIC

See also

- `Base (Member Access)`
- `This`

- **Type**
- **Extends**
- **Option Base**

Base (Member Access)



Provides explicit access to base type members in non-static methods of

Syntax

```
Base .member  
Base [ .Base ... ] .member
```

Description

Base provides a way to explicitly access members of a specific base type of a user-defined type derived from another type using **Extends**.

By using **Base** repeatedly, as in `base.base.base.member`, it is possible to handle there are multiple levels of inheritance.

Base is especially useful when a base type's member is shadowed by a derived type using the same identifier. **Base** then allows unambiguous access to the base member.

For virtual methods, `base.method()` always calls the base method and not the derived method.

Example

```
Type Parent  
  As Integer a  
  Declare Constructor(ByVal As Integer = 0)  
  Declare Sub show()  
End Type  
  
Constructor Parent(ByVal a As Integer = 0)  
  This.a = a  
End Constructor  
  
Sub Parent.show()  
  Print "parent", a  
End Sub  
  
Type Child extends Parent
```

```

    As Integer a
    Declare Constructor(ByVal As Integer = 0)
    Declare Sub show()
End Type

Constructor Child(ByVal a As Integer = 0)
    ' Call base type's constructor
    Base(a * 3)
    This.a = a
End Constructor

Sub Child.show()
    ' Call base type's show() method, not ours
    Base.show()

    ' Show both a fields, the base type's and our
    Print "child", Base.a, a
End Sub

Type GrandChild extends Child
    As Integer a
    Declare Constructor(ByVal As Integer = 0)
    Declare Sub show()
End Type

Constructor GrandChild(ByVal a As Integer = 0)
    ' Call base type's constructor
    Base(a * 2)
    This.a = a
End Constructor

Sub GrandChild.show()
    ' Call base type's show() method, not ours
    Base.show()

    ' Show both a fields, the base.base type's, t
    Print "grandchild", Base.Base.a, Base.a, a
End Sub

```

```
Dim As GrandChild x = GrandChild(3)
x.show()
```

Dialect Differences

- Methods are only supported in the *-lang fb* dialect, hence **Base**

Differences from QB

- New to FreeBASIC

See also

- **Base (Initializer)**
- **This**
- **Type**
- **Extends**
- **Option Base**

Beep



Produces a beep sound.

Syntax

```
Declare Sub Beep ( )
```

Usage

```
Beep
```

Description

Beep tells the system to sound a beep noise. Note that this might not work on some platforms. Since this command is not reliable and there is no way to specify the frequency and duration, you might want to avoid it in favor of other / better solutions, for example: <http://www.freebasic.net/forum/viewtopic.php?p=20441#20441> by yetifoot.

Example

```
Beep
```

Differences from QB

- In QB, this was a single tone noise generated through the PC speaker. Now this might not be the case.

See also

- `out` - producing sound using CPU ports

Returns a binary (base 2) string representation of an integer

Syntax

```
Declare Function Bin ( ByVal number As UByte ) As String
Declare Function Bin ( ByVal number As UShort ) As String
Declare Function Bin ( ByVal number As ULong ) As String
Declare Function Bin ( ByVal number As ULongInt ) As String
Declare Function Bin ( ByVal number As Const Any Ptr ) As String

Declare Function Bin ( ByVal number As UByte, ByVal digits As
Long ) As String
Declare Function Bin ( ByVal number As UShort, ByVal digits As
Long ) As String
Declare Function Bin ( ByVal number As ULong, ByVal digits As
Long ) As String
Declare Function Bin ( ByVal number As ULongInt, ByVal digits As
Long ) As String
Declare Function Bin ( ByVal number As Const Any Ptr, ByVal
digits As Long ) As String
```

Usage

```
result = Bin[$]( number [, digits ] )
```

Parameters

number

A number or expression evaluating to a number. A floating-point number will be converted to a **LongInt**.

digits

Desired number of digits in the returned string.

Return Value

A string containing the unsigned binary representation of *number*.

Description

Returns a string representing the unsigned binary value of the integer *number*. Binary digits range from 0 to 1.

If you specify *digits* > 0, the result string will be exactly that length. It will be truncated or padded with zeros on the left, if necessary.

The length of the string will not go longer than the maximum number of digits required for the type of *number* (32 for a **Long**, 64 for a **LongInt**).

If you want to do the opposite, i.e. convert a binary string back into a number, the easiest way to do it is to prepend the string with "&B;", and convert it to an integer type, using a function like **cInt**, similarly to a normal numeric string. E.g. **cInt("&B101;")**

Example

```
Print Bin(54321)
Print Bin(54321, 5)
Print Bin(54321, 20)
```

will produce the output:

```
1101010000110001
10001
00001101010000110001
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias **__Bin**.

Differences from QB

- New to FreeBASIC

See also

- **Oct**

- Hex
- ValInt
- ValLng

Specifies file or device to be opened for binary mode

Syntax

`Open filename for Binary [Access access_type] [Lock lock_type] a`

Parameters

filename

file name to open

access_type

indicates whether the file may be read from, written to or both

lock_type

locking to be used while the file is open

filenum

unused file number to associate with the open file

Description

Opens a file or device for reading and/or writing binary data in the file. If the file does not exist, a new file will be created. The file pointer is in `Get #` and `Put #` file operations move the file pointer according to the `s` byte in the file.

The data existing in the file is preserved by `Open`.

This file mode can use any buffer variable to read/write data in the file. The data is saved in binary mode, in the same internal format FreeBA

filename must be a string expression resulting in a legal file name in `t` be sought for in the present directory, unless a path is given.

Access_type By default `Binary` mode allows to both read and write the be one of:

- `Read` - the file is opened for input only
- `Write` - the file is opened for output only
- `Read Write` - the file is opened for input and output (the `o`

Lock_type indicates the way the file is locked for other processes (use

- `Shared` - The file can be freely accessed by other proces

- **Lock Read** - The file can't be opened simultaneously for
- **Lock Write** - The file can't be opened simultaneously for
- **Lock Read Write** - The file cannot be opened simultaneously

If no lock type is stated, the file will be **shared** for other threads of the program.

Lock and **Unlock** can be used to restrict temporarily access to parts of a

filenum is a valid file number (in the range 1..255) not being used for other files. It identifies the file for the rest of file operations. A free file number can be

Example

```
' ' Create a binary data file with one number in it
Dim x As Single = 17.164

Open "MyFile.Dat" For Binary As #1
' ' put without a position setting will put from
' ' in this case, the very beginning of the file.
Put #1, , x
Close #1
```

```
' ' Now read the number from the file
Dim x As Single = 0

Open "MyFile.Dat" For Binary As #1
Get #1, , x
Close #1

Print x
```

```
' ' Read entire contents of a file to a string
Dim txt As String
```

```
Open "myfile.txt" For Binary Access Read As #1
  If LOF(1) > 0 Then
    ' our string has as many characters as the fi
    txt = String(LOF(1), 0)
    ' size of txt is known. entire string filled
    Get #1, , txt
  End If
Close #1

Print txt
```

Differences from QB

- None

See also

- Open
- Put #
- Get #
- Random
- Append
- Output
- Input

Gets the state of an individual bit in an integer value.

Syntax

```
#define Bit( value, bit_number ) (((value) And  
(Cast(.TypeOf(value), 1) Shl (bit_number))) <> 0)
```

Usage

```
result = Bit( value, bit_number )
```

Parameters

value

The integer value.

bit_number

The index of the bit.

Return Value

Returns an **Integer** value of -1 if the bit is set, or 0 if the bit is cleared.

Description

This macro expands to an integer value indicating whether or not the bit specified by *bit_number* is set in the integer *value*. Behaves as ``(value And 1 Shl bit_number) <> 0``.

Example

```
Print Bit(4,2)  
Print Bit(5,1)  
Print Bit(&H8000000000000000ULL, 63)
```

will produce the output:

```
-1  
0  
-1
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__Bit`.

Differences from QB

- New to FreeBASIC

See also

- `BitSet`
- `BitReset`

BitReset



Gets the value of an integer with a specified bit cleared.

Syntax

```
#define BitReset( value, bit_number ) ((value) And Not  
  (Cast(.TypeOf(Value), 1) Shl (bit_number)))
```

Usage

```
result = BitReset( value, bit_number )
```

Parameters

value

The integer value.

bit_number

The index of the bit to clear.

Return Value

Returns the integer value with the specified bit cleared.

Description

This macro expands to a copy of the integer *value* with the specified *bit_number* cleared (to off, or `0`). Behaves as ``value And Not (1 Shl bit_number)``.

The valid range of values for *bit_number* depends on the size, in bits, of ``TypeOf(value)``, which is `0` through ``sizeof(value) * 8 - 1``. See [Standard Datatype Limits](#) for a table of the standard datatypes and their sizes.

Example

```
Print BitReset(5,0)  
Print Hex(BitReset(&h800000000000000001,63))
```

will produce the output:

```
4  
1
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__Bitreset`.

Differences from QB

- New to FreeBASIC.

See also

- [Bit](#)
- [BitSet](#)

Gets the value of an integer with a specified bit set.

Syntax

```
#define BitSet( value, bit_number ) ((value) Or  
(Cast(.TypeOf(Value), 1) Shl (bit_number)))
```

Usage

```
result = BitSet( value, bit_number )
```

Parameters

value

The integer value.

bit_number

The index of the bit to set.

Return Value

Returns the integer value with the specified bit set.

Description

This macro expands to a copy of the integer *value* with the specified *bit_number* set (to *on*, or ``1``). Behaves as ``value Or (1 Shl bit_number)``.

The valid range of values for *bit_number* depends on the size, in bits, of ``TypeOf(value)``, which is ``0`` through ``sizeof(value) * 8 - 1``. See [Standard Datatype Limits](#) for a table of the standard datatypes and their sizes.

Example

```
Print BitSet(4, 0)  
Print Hex(BitSet(1ull, 63))
```

will produce the output:

```
5  
800000000000000001
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__Bitset`.

Differences from QB

- New to FreeBASIC.

See also

- [Bit](#)
- [BitReset](#)

BLoad



Loads arbitrary data from a file created with **BSave**, or a compatible BMP

Syntax

```
Declare Function BLoad ( ByRef filename As Const String, ByVal d
) As Long
```

Usage

```
result = BLoad( filename [, [ dest ] [, pal ] ] )
```

Parameters

filename

the name of the file to load the image from; can include a file path

dest

the memory location to load the image to, or null (0) to copy the image

pal

the memory location to load the palette to, or null (0) to change the cu

Return Value

Returns zero (0) if successful, or a non-zero error code to indicate a fa

Description

BLoad can be used to load image data or any other data from a file cre or paste it to the screen. If *dest* is absent or null (0), the image data is page. Otherwise it is loaded as image data to the address given by *de*

BLoad can load 3 different types of files:

- Old QB-like data files, saved with **BSAVE** from QB code, (header, beginning with &HFD; , up to 64 KiB in size
- New FB-like data files, saved with **BSave** from FB code, (header, beginning with &HFE; . There is no 64 KiB limit w
- BMP image files, supports a subset of valid ("Windows" code with **BSave**, or created / saved in a compatible form

QB-like data files and BMP files are converted to an FB-compatible im

Image files with 8-bit per pixel resolution or lower contain a palette that is used to map colors to the screen. If `pal` is not null (`0`), the palette is copied to memory starting at the address `pal`. If the graphics screen uses a palette then its palette is changed to match that of the image.

When using one of the 2 "non-BMP" file formats to save images, the image is saved in the same graphics screen mode as it is being loaded into. When using a palette, the palette is copied to memory and applied to the screen.

When loading a BMP file using `BLoad`, the images must be true-color (24 bits per pixel) or palettized/indexed (8-bit or lower). The image data will be converted to the depth of the image buffer, except that true-color can't be reduced to a palettized image. `BLoad` does not support compression or other image file types (PNG, JPG, GIF, ...). `BLoad` will not load 32-bit BMP files with `BITMAPV4HEADER` or `BITMAPV5HEADER` file headers.

Runtime errors:

`BLoad` throws one of the following **runtime errors**:

(1) *Illegal function call*

- `dest` was not specified or was null (`0`), and no graphics screen was open.
- The Bitmap uses an unsupported BMP file compression.
- The Bitmap is true-color (16, 24, or 32 bits per pixel) and the image buffer is not 16, 24, or 32 bits per pixel or lower).

(2) *File not found*

- The file `filename` could not be found.

(3) *File I/O error*

- The file doesn't have any of the supported types.
- A general read error occurred.

Note: When you use `BLoad` to load a BMP file into an image buffer, the dimensions of the image buffer are changed. If you want the image buffer to have the same dimensions as the image, you must specify the dimensions beforehand, and create an image of the right size yourself. See [Image Buffer](#) for how to do this.

Example

```
'Load a graphic to current work page
```

```
Screen 18, 32
Cls
BLoad "picture.bmp"
Sleep
```

```
'Load a 48x48 bitmap into an image
ScreenRes 320, 200, 32
Dim myImage As Any Ptr = ImageCreate( 48, 48 )
BLoad "picture.bmp", myImage
Put (10,10), myImage
ImageDestroy( myImage )
Sleep
```

```
ScreenRes 640, 480, 8 ' 8-bit palette graphics mode
Dim pal(0 To 256-1) As Integer ' 32-bit integer array

' load bitmap to screen, put palette into pal() array
BLoad "picture.bmp", , @pal(0)

WindowTitle "Old palette"
Sleep

' set new palette from pal() array
Palette Using pal(0)

WindowTitle "New palette"
Sleep
```

```
' A function that creates an image buffer with the
' dimensions as a BMP image, and loads a file into it
```

```

Const NULL As Any Ptr = 0

Function bmp_load( ByRef filename As Const String

    Dim As Long filenum, bmpwidth, bmpheight
    Dim As Any Ptr img

    ' open BMP file
    filenum = FreeFile()
    If Open( filename For Binary Access Read As #f

        ' retrieve BMP dimensions
        Get #filenum, 19, bmpwidth
        Get #filenum, 23, bmpheight

    Close #filenum

    ' create image with BMP dimensions
    img = ImageCreate( bmpwidth, Abs(bmpheight) )

    If img = NULL Then Return NULL

    ' load BMP file into image buffer
    If BLoad( filename, img ) <> 0 Then ImageDestr

    Return img

End Function

Dim As Any Ptr img

ScreenRes 640, 480, 32

img = bmp_load( "picture.bmp" )

If img = NULL Then

```

```
Print "bmp_load failed"  
  
Else  
  
Put (10, 10), img  
  
ImageDestroy( img )  
  
End If  
  
Sleep
```

Differences from QB

- Support for loading BMP files is new to FreeBASIC.
- Support for retrieving the palette from BMP files is new to Free
- FreeBASIC uses a different file format from QBASIC internally, unsupported by QBASIC.

See also

- **BSave**
- **Palette**
- **ImageCreate**
- **ImageDestroy**
- **Internal Graphics Formats**

Standard data type

Syntax

```
Dim variable As Boolean
```

Description

Boolean data type. Can hold the values **True** or **False**.

Notes on definition of boolean data type: *Ideally, the definition of the boolean data type is that it holds the value of **True** or **False**, and that's it. However, to make this concept a reality, we need a definition that uses real world connections. A more realistic definition is that the boolean data type is a 1-bit integer, having the value 0 to indicate **False** and 1 to indicate **True**. For a practical definition, we must consider, yet again, additional factors. The most significant factor is that the hardware (processor) on which code is executed does not directly support a 1-bit data type; the smallest register or memory size we can work with is 8-bits or 1-byte. Therefore, a practical definition of boolean data type is an integer, 8 bits wide, having the value 0 or 1, where all other values are undefined. However, because of longstanding differences between C/C++ and FB with respect to logic operations, the interpretation of the value must also be considered. Assume "false" is 0 in both C/C++ and FB. C/C++ has logical 'not' operator '!' such that '!0' produces '1'. FB has a bitwise **Not** operator such that 'not 0' produces '-1'. Therefore the definition for a C/C++ boolean is an unsigned 1-bit integer, zero extended to fill larger integer types, and the definition for a FB boolean is a signed 1-bit integer, sign extended to fill larger integer types. However, the purpose and intent of the boolean data type remains, that it should only ever hold a **True** value or **False** value, regardless of the underlying details.*

Example

```
Dim boolvar As Boolean
```

```
boolvar = True
Print "boolvar = ", boolvar
```

Output:

```
boolvar =      true
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__Boolean`.

Differences from QB

- New to FreeBASIC

See also

- [True](#)
- [False](#)

Saves an array of arbitrary data and palette information to a file on disk

Syntax

```
Declare Function BSave ( ByRef filename As Const String, ByVal  
source As Any Ptr, ByVal size As ULong = 0, ByVal pal As Any Ptr  
= 0, ByVal bitsperpixel As Long = 0 ) As Long
```

Usage

```
result = BSave( filename, source [, [ size ] [, [ pal ] [,  
bitsperpixel ]]] )
```

Parameters

filename

the name of the file to create for storing the pixel and palette data.

source

the address of the data to store, or null (0) to store pixel data from the current screen work page.

size

optional, the total number of bytes of data to store. This value is needed unless the output is a BMP file.

pal

optional, the address of a buffer holding 256 **palette** colors, or null (0) for the current screen palette.

bitsperpixel

optional, a requested bit depth for the output BMP image.

Return Value

Returns zero (0) if successful, or a non-zero error code to indicate a failure. (*throws a runtime error*)

Description

Bsave is used for saving arbitrary data from memory into a file, using a file format specific to FB, or saving images into a standard BMP image file, replacing an existing file if necessary.

BSave outputs a total of *size* bytes of arbitrary data located at *source* to a specified file. If *source* is null (0), then **BSave** outputs a maximum of *size* bytes from the current work page's pixel buffer, which is structure in the current screen mode's internal pixel format. (This data is not compatible with the image buffer format as it has no header.) For 8-bit images, palette information is obtained from *pal* if present and non-null, or if *pal* omitted or null (0), from the current screen palette.

A BMP image file can be created if *filename* has a file extension of ".bmp" (case insensitive). *source* is assumed to point to a valid image buffer whose entire pixel data will be stored in the BMP file. If *source* is null (0), the contents of the current work page will be stored instead. For 8-bit images, palette information is obtained from *pal* if non-null, or if null (0), from the current screen palette. The *size* parameter is ignored when saving BMP files.

The default bit depth for BMP files is 8-bit for 8-bit (palette) images, 24-bit for 16-bit images, and 32-bit for 32-bit images. The *bitsperpixel* parameter can be used to request 24-bit output for 8-bit images, or 24-bit output for 32-bit images.

Runtime errors:

BSave throws one of the following **runtime errors**:

(1) *Illegal function call*

- *size* is less than zero (0), or *size* is zero and *source* is non-null, or a problem is detected with the image buffer.

(2) *File not found*

- The file could not be created.

(3) *File I/O error*

- The file could not be written to.

Example

```
' Set gfx mode  
ScreenRes 320, 200, 32
```

```
' Clear with black on white
Color RGB(0, 0, 0), RGB(255, 255, 255)
Cls

Locate 13, 15: Print "Hello world!"

' Save as BMP
BSave "hello.bmp", 0
```

Differences from QB

- Support for saving more than 64KiB of arbitrary data is new to FreeBASIC.
- Support for saving BMP files is new to FreeBASIC.
- QB cannot use **BLoad** to load files created with **BSave** in FreeBASIC, but FreeBASIC can use **BLoad** to load files created with **BSave** in QB

See also

- **BLoad**
- **Palette**

Byref (Parameters)



Declaration specifier to explicitly pass a parameter by reference

Syntax

```
ByRef param As datatype
```

Usage

```
[ Declare ] { Sub | Function } proc_name ( ByRef param As datatype )
```

Description

Passes a variable by reference, that is its address, to a subroutine or function. When a variable is passed by reference, the contents of the variable can be changed by the target subroutine or function.

In *-lang qb* and *-lang fblite* dialects, **ByRef** is the default parameter passing convention, unless **Option ByVal** is in effect.

Opposite of **ByVal**.

Example

```
Dim MyVar As Integer

Sub ChangeVar(ByRef AVar As Integer)
    AVar = AVar + 1
End Sub

MyVar = 1
Print "MyVar: "; MyVar 'output = 1
ChangeVar MyVar
Print "MyVar: "; MyVar 'output = 2
Sleep
End
```

Dialect Differences

- In *-lang fb* dialect, `ByVal` is the default parameter passing convention for all built-in types except `string` and user-defined `Type` which are passed `ByRef` by default.
- In *-lang qb* and *-lang fblite* dialects, `ByRef` is the default parameter passing convention.

Differences from QB

- New to FreeBASIC

See also

- [Passing Arguments to Procedures](#)
- [Declare](#)
- [ByVal](#)
- [Byref \(Function Results\)](#)

Byref (Function Results)



Specifies that a function result is returned by reference

Syntax

```
Function name ( parameter-list ) ByRef As datatype
```

Description

Causes the function result to be returned by reference, rather than by returning by value. This allows the caller of the function to modify the

If **ByRef** is not specified, the default is to return the function result by v

Functions with **ByRef** result should not return local variables from the f reference to them. To help with writing safe code, the compiler will sho x statements.

Note: On the left-hand side of an assignment expression using the '=' function calls one single argument, in order to solve the parsing ambigu allowing to avoid parsing ambiguity (without parentheses). As for the :

Operators (member or global), when used as functions, have also the

Example

```
Function min( ByRef I As Integer , ByRef J As Integer )
    ' The smallest integer will be returned by reference
    If I < J Then
        Return I
    Else
        Return J
    End If
End Function

Dim As Integer A = 13, B = 7
Print A, B
```

```
Print min( A , B )
min( A , B ) = 0
Print A, B
```

```
Function f( ) ByRef As Const ZString
    ' This string literal (because statically all
    Function = "abcd"
End Function

Print f( )
```

```
Dim Shared As String s

Function f1( ) ByRef As String
    ' This variable-length string will be returned
    Function = s
End Function

Function f2( ByRef _s As String ) ByRef As String
    ' This variable-length string will be returned
    Function = _s
End Function

s = "abcd"
Print s

f1( ) &= "efgh"
Print s

' At time of writing, the enclosing parentheses are
( f2( s ) ) &= "ijkl"
Print s
```

```
Function power2( ByRef _I As Integer ) ByRef As Integer
```

```
_I *= _I
'' This integer will be returned by reference,
Function = _I
End Function

Dim As Integer I = 2
power2( power2( power2( I ) ) ) '' Function retur
Print I
```

Differences from QB

- New to FreeBASIC

See also

- [Returning values](#)
- [Byref \(Parameters\)](#)

Byte



Standard data type: 8 bit signed

Syntax

```
Dim variable As Byte
```

Description

8-bit signed whole-number data type. Can hold a value in the range of -128 to 127.

Example

```
Dim bytevar As Byte
bytevar = 100
Print "bytevar= ", bytevar
```

```
Dim x As Byte = CByte(&H80)
Dim y As Byte = CByte(&H7F)
Print "Byte Range = "; x; " to "; y
```

Output:

```
Byte Range = -128 to 127
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__Byte`.

Differences from QB

- New to FreeBASIC

See also

- **UByte**
- **CByte**

Declaration specifier to explicitly pass a parameter by value

Syntax

ByVal *param* **As** *datatype*

Usage

[**Declare**] { **Sub** | **Function** } *proc_name* (**ByVal** *param* **As** *datatype*

Description

ByVal in a parameter list of a declare statement causes a copy of the variable to be passed, not its value.

This means that if the value of the variable *x* is passed, then the original variable *x* were passed **ByRef**, the value of the original variable *x* could be modified.

Opposite of **ByRef**.

The **ByVal** keyword is also used in the context of **Byref Parameters** at reference semantics in order to pass or assign a pointer as-is to a **ByRef** parameter.

- **Manually passing pointers to by-reference parameters**
- **Manually returning pointers as-is from Byref functions**

Example

```
Sub MySub(ByVal value As Integer)
    value += 1
End Sub

Dim MyVar As Integer

MyVar = 1
Print "MyVar: "; MyVar 'output = 1
MySub MyVar
```

```
Print "MyVar: "; MyVar 'output = 1, because byval  
Sleep  
End
```

Dialect Differences

- In the *-lang fb* dialect, **ByVal** is the default parameter passing c are passed **ByRef** by default.
- In *-lang qb* and *-lang fblite* dialects, **ByRef** is the default paran

Differences from QB

- QB only used **ByVal** in declarations to non-Basic subroutines

See also

- **Passing Arguments to Procedures**
- **Declare**
- **ByRef**

Call



Statement to invoke a subroutine

Syntax

```
Call procname ([parameter list])
```

Description

Calls a **Sub** or **Function**.

This keyword is a holdover from earlier dialects of BASIC, and is mainly

In **-lang qb**, it can be used to call **subs** in code before they are declared. Parameters passed **ByRef** As **Any**.

Note: until the function is declared, no type-checking is done on the parameters; they are of the correct type.

Example

```
' ' Compile with -lang qb or -lang fblite
#lang "fblite"

Declare Sub foobar(ByVal x As Integer, ByVal y As Integer)
Call foobar(35, 42)

Sub foobar(ByVal x As Integer, ByVal y As Integer)
Print x; y
End Sub
```

```
' ' Compile with -lang qb or -lang fblite
#lang "fblite"
```

```
Function f ( ) As Integer
```

```
f = 42
```

```
End Function
```

```
Call f ' execute function f, but ignore the answer
```

```
' ' Compile with -lang qb
```

```
'$lang: "qb"
```

```
Call mysub(15, 16) ' call "mysub" before it has been declared
```

```
Sub mysub(ByRef a As Integer, ByRef b As Integer)
```

```
Print a, b
```

```
End Sub
```

Dialect Differences

- The use of `call` is not allowed in the *-lang fb* dialect.
- The *-lang fblite* dialect does not allow you to call functions that have not been declared.

Differences from QB

- The procedure must have already been declared.
- `call` in QB will make a copy of all parameters, so changes made in the procedure are not reflected in the variables in the caller.

See also

- [Declare](#)
- [Sub](#)

CAllocate



Allocates memory for a certain number of elements from the free store and returns the address of the first element. The contents of the allocated memory are uninitialized.

Syntax

```
Declare Function CAllocate cdecl ( ByVal num_elements As UInteger  
As UInteger = 1 ) As Any Ptr
```

Usage

```
result = CAllocate( num_elements [, size ] )
```

Parameters

num_elements

The number of elements to allocate memory for.

size

The size, in bytes, of each element.

Return Value

If successful, the address of the start of the allocated memory is returned. If the allocation fails, the null pointer (0) is returned.

Description

CAllocate initializes the allocated memory with zeros. Consequently, CAllocate can be also directly used with **string** or **udt** containing string, because the descriptor is cleared (set to 0) first.

Example

```
' Allocate and initialize space for 10 integer elements.  
Dim p As Integer Ptr = CAllocate(10, SizeOf(Integer))  
  
' Fill the memory with integer values.  
For index As Integer = 0 To 9  
    p[index] = (index + 1) * 10
```

```
Next
' Display the integer values.
For index As Integer = 0 To 9
    Print p[index] ;
Next

' Free the memory.
Deallocate(p)
```

Outputs:

```
10 20 30 40 50 60 70 80 90 100
```

Platform Differences

- This procedure is not guaranteed to be thread-safe.

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the `__Callocate`.

Differences from QB

- New to FreeBASIC

See also

- [Allocate](#)
- [Deallocate](#)
- [Reallocate](#)

Control flow statement

Syntax

`Case` *expression*

Differences from QB

- None

See also

- [Select Case](#)

Cast



Converts an expression to a specified data type

Syntax

```
Cast( datatype, expression )
```

Description

Converts *expression* into a different **datatype**. Useful to be used in macro converting to **Type Alias**.

Note: this is a general form of conversion operators such as **cInt** or **ci** used on types that have a **cast operator**, but don't have a built-in key also suitable for use in cases where the type of a variable is not fixed earlier, or may be the **Type of** a different variable or expression.

Note: If you want to use an operator specifically for converting to different instead.

Example

```
' ' will print -128 because the integer literal will  
' ' (this Casting operation is equivalent to using  
Print Cast( Byte, &h0080 )  
  
' ' will print 3 because the floating-point value will  
' ' (this Casting operator is equivalent to using C  
Print Cast( Integer, 3.1 )
```

Dialect Differences

- Not available in the **-lang qb** dialect unless referenced with the

Differences from QB

- [New to FreeBASIC](#)

See also

- [CPtr](#)
- [CInt](#)
- [TypeOf](#)

Converts numeric or string expression to a boolean (**Boolean**)

Syntax

```
Declare Function Cbool ( ByVal expression As datatype ) As Boolean
```

```
Type typename
```

```
Declare Operator Cast ( ) As Boolean
```

```
End Type
```

Usage

```
result = Cbool( numeric expression )
```

```
result = Cbool( string expression )
```

```
result = Cbool( user defined type )
```

Parameters

expression

a numeric, string, or user defined type to cast to a **Boolean** value

datatype

any numeric, string, or user defined type

typename

a user defined type

Return Value

A **Boolean** value.

Description

The **cbool** function converts a zero value to **False** and a non-zero value to **True**.

The name can be explained as 'Convert to Boolean'.

If the argument is a string expression, it is converted to boolean using insensitive to the string "false" to return a **False** value or "true" to return a **True** value.

Example

```
' Using the CBOOL function to convert a numeric va  
  
'Create an BOOLEAN variable  
Dim b As BOOLEAN  
  
'Convert a numeric value  
b = CBOOL(1)  
  
'Print the result, should return True  
Print b  
Sleep
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the `__cbool`.

Differences from QB

- New to FreeBASIC

See also

- CByte
- CByte
- CShort
- CUShort
- CInt
- CUInt
- CLng
- CULng
- CLngInt

- **CULngInt**
- **CSng**
- **Cdbl**
- **Str**

Converts numeric or string expression to **Byte**.

Syntax

```
Declare Function CByte ( ByVal expression As datatype ) As Byte
```

```
Type typename
```

```
Declare Operator Cast ( ) As Byte
```

```
End Type
```

Usage

```
result = CByte( numeric expression )
```

```
result = CByte( string expression )
```

```
result = CByte( user defined type )
```

Parameters

expression

A numeric, string, or pointer expression to cast to a **Byte** value.

datatype

Any numeric, string, or pointer data type.

typename

A user defined type.

Return Value

A **Byte** value.

Description

The **cByte** function rounds off the decimal part and returns a 8-bit **Byte** function does not check for an overflow, and results are undefined for which are less than -128 or larger than 127.

The name can be explained as 'Convert to Byte'.

If the argument is a string expression, it is converted to numeric by us

Example

```
' Using the CBYTE function to convert a numeric va  
  
'Create an BYTE variable  
Dim numeric_value As Byte  
  
'Convert a numeric value  
numeric_value = CByte(-66.30)  
  
'Print the result, should return -66  
Print numeric_value  
Sleep
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the `__cbyte`.

Differences from QB

- New to FreeBASIC

See also

- `CByte`
- `CShort`
- `CUShort`
- `CInt`
- `CUInt`
- `CLng`
- `CULng`
- `CLngInt`
- `CULngInt`

- CSng
- CDb1

Converts numeric or string expression to **Double** precision floating point

Syntax

```
Declare Function Cdbl ( ByVal expression As datatype ) As Double
```

```
Type typename
```

```
Declare Operator Cast ( ) As Double
```

```
End Type
```

Usage

```
result = Cdbl( numeric expression )
```

```
result = Cdbl( string expression )
```

```
result = Cdbl( user defined type )
```

Parameters

expression

a numeric, string, or pointer expression to cast to a **Double** value

datatype

any numeric, string, or pointer data type

typename

a user defined type

Return Value

A **Double** precision value.

Description

The **cdbl** function returns a 64-bit **Double** value. The function does not an overflow, so be sure not to pass a value outside the representable the **Double** data type. The name can be explained as 'Convert to DouE

If the argument to **cdbl** is a string expression, it is first converted to nu using **Val**.

Example

```
' Using the CDBL function to convert a numeric val  
  
'Create an DOUBLE variable  
Dim numeric_value As Double  
  
'Convert a numeric value  
numeric_value = CDb1(-12345678.123)  
  
'Print the result, should return -12345678.123  
Print numeric_value  
Sleep
```

Differences from QB

- The string argument was not allowed in QB

See also

- CByte
- CByte
- CShort
- CUShort
- CInt
- CUInt
- CLng
- CULng
- CLngInt
- CULngInt
- CSng

Specifies a *cdecl*-style calling convention in a procedure declaration

Syntax

```
Sub name cdecl [Overload] [Alias "alias"] ( parameters )  
Function name cdecl [Overload] [Alias "alias"] ( parameters ) As
```

Description

In procedure declarations, `cdecl` specifies that a procedure will use the stack (pushed onto the stack) in the reverse order in which they are listed, and must not clean up the stack (pop any parameters) before it returns - the

`cdecl` is allowed to be used with variadic procedure declarations (those

`cdecl` is the default calling convention on Linux, the *BSDs, and DOS, **Blocks**. `cdecl` is typically the default calling convention for C compilers

Example

```
' declaring 'strcpy' from the standard C library  
Declare Function strcpy cdecl Alias "strcpy" (ByVa
```

Differences from QB

- New to FreeBASIC

See also

- `pascal`, `stdcall`
- `Declare`
- `Sub`, `Function`

Chain



Temporarily transfers control to an external program

Syntax

```
Declare Function Chain ( ByRef program As Const String ) As Long
```

Usage

```
result = Chain( program )
```

Parameters

program

The file name (including file path) of the program (executable) to transfer control to.

Return Value

Returns the external program's exit code if executed successfully, or negative one (-1) otherwise.

Description

Transfers control over to an external program. When the program exits, execution resumes immediately after the call to **chain**.

Example

```
#ifdef __FB_LINUX__
    Dim As String program = "./program"
#else
    Dim As String program = "program.exe"
#endif

Print "Running " & program & "... "
If (Chain(program) <> 0) Then
    Print program & " not found!"
```

End If

Platform Differences

- Linux requires the *program* name case matches the real name of the file. Windows and DOS are case insensitive. The program chained may be case sensitive for its command line parameters.
- Path separators in Linux are forward slashes / . Windows uses backward slashes \ but it allows for forward slashes . DOS use backward \ slashes.
- Exit code is limited to 8 bits in DOS.

Differences from QB

- None

See also

- **Exec** transfer temporarily, with arguments
- **Run** one-way transfer
- **Command** pick arguments

ChDir



Changes the current drive and directory

Syntax

```
Declare Function ChDir ( ByRef path As Const String ) As Long
```

Usage

```
result = ChDir( path )
```

Parameters

path

A **String** argument specifying the path to change to.

Return Value

Returns zero (0) on success and negative one (-1) on failure.

Description

Changes the current drive and directory to that specified.

Example

```
Dim pathname As String = "x:\folder"  
Dim result As Integer = ChDir(pathname)  
  
If 0 <> result Then Print "error changing current"
```

Platform Differences

- Linux requires the *filename* case matches the real name of the
- Path separators in Linux are forward slashes / . Windows uses slashes . DOS uses backward \ slashes.

Differences from QB

- In QB, the drive could not be specified.

See also

- `MkDir`
- `Rmdir`

Returns a string of characters from one or more ASCII integer values

Syntax

```
Declare Function Chr ( ByVal ch As Integer [, ... ] ) As String
```

Usage

```
result = Chr[$]( ch0 [, ch1 ... chN ] )
```

Parameters

ch

The **ASCII** integer value of a character.

Return Value

Returns a string containing the character(s).

Description

chr returns a string containing the character(s) represented by the **ASCII** values passed to it.

When **chr** is used with numerical constants or literals, the result is evaluated at compile-time, so it can be used in variable initializers.

Asc performs the opposite function, returning the **ASCII** code of a character represented by a string.

Example

```
Print "the character represented by";  
Print " the ASCII code of 97 is: "; Chr(97)  
  
Print Chr(97, 98, 99) ' prints abc
```

```
' s initially has the value "abc"  
Dim s As String = Chr(97, 98, 99)  
  
Print s
```

Dialect Differences

- The string type suffix "\$" is obligatory in the *-lang qb* dialect.
- The string type suffix "\$" is optional in the *-lang fblite* and *-lang fb* dialects.

Differences from QB

- FreeBASIC accepts multiple integer values as arguments, QB accepted only one.
- FreeBASIC evaluates the CHR function at compile time when used with constants or literals.

See also

- **ASCII Character Codes**
- **Asc**
- **Str**
- **Val**

Converts a numeric or string expression to an **Integer** or an **Integer**<*bits*>

Syntax

```
Declare Function CInt ( ByVal expression As datatype ) As Integer
Declare Function CInt<bits> ( ByVal expression As datatype ) As
```

```
Type typename
```

```
Declare Operator Cast ( ) As Integer
```

```
Declare Operator Cast ( ) As Integer<bits>
```

```
End Type
```

Usage

```
result = CInt( expression )
result = CInt( string expression )
result = CInt( user defined type )
```

Parameters

bits

A numeric constant expression indicating the size in bits of integer declared. Allowed are 8, 16, 32 or 64.

expression

a numeric, string, or pointer expression to cast to a **Integer** value

datatype

any numeric, string, or pointer data type

typename

a user defined type

Return Value

An **Integer** or **Integer**<*bits*> containing the converted value.

Description

If **CInt** is passed a numeric *expression*, it rounds it using the **Round** method - i.e. it rounds to the closest integer value, choosing the closest number if the number is equidistant from two integers - and returns an **Integer**. If *bits* is supplied, an integer type of the given size.

The function does not check for an overflow; for example, for a 32-bit results are undefined for values which are less than $-2\ 147\ 483\ 648$ or $2\ 147\ 483\ 647$.

If the argument is a string expression, it is converted to numeric by us `ValLng`, depending on the size of the result type.

The name "CINT" is derived from 'Convert to INTEger'.

Example

```
' Using the CINT function to convert a numeric val
'Create an INTEGER variable
Dim numeric_value As Integer
'Convert a numeric value
numeric_value = CInt(300.5)
'Print the result, should return 300, because 300
numeric_value = CInt(301.5)
'Print the result, should return 302, because 301
Print numeric_value
```

Dialect Differences

- In the *-lang qb* dialect, `cInt` will return a 16-bit integer, like in C

Differences from QB

- The string argument was not allowed in QB
- The *<bits>* parameter was not allowed in QB

See also

- **Cast**
- **CByte**
- **CByte**
- **CShort**
- **CUShort**
- **CUInt**
- **CLng**
- **CULng**
- **CLngInt**
- **CULngInt**
- **CSng**
- **Cdbl**
- **Integer**

Circle



Graphics statement to draw an ellipse or a circle

Syntax

```
circle [target,] [STEP] (x,y), radius[, [color][, [start][, [end  
[aspect][, F]]]]]
```

Parameters

target

optional; specifies the image buffer to draw on

STEP

indicates that coordinates are relative

(*x*, *y*)

coordinates of the center of the ellipse

radius

the radius of the circle - or for an ellipse, the semi-major axis (i.e. the longest radius)

color

the color attribute

start

starting angle

end

ending angle

aspect

aspect ratio of the ellipse, the ratio of the height to the width

F

fill mode indicator

Description

circle will draw a circle, ellipse, or arc based on the parameters given

target specifies a buffer to draw on. *target* may be an image created **ImageCreate** or **Get (Graphics)**. If omitted, *target* defaults to the screen current work page. (See **ScreenSet**)

The center of the shape will be placed on the destination surface at (>

Radius denotes the radius of the shape. If *aspect* ratio is not 1.0, the *radius* must be given here.

color denotes the color attribute, which is mode specific (see **color** at **Screen (Graphics)** for details). If omitted, the current foreground color by the **color** statement is used.

The **step** option specifies that *x* and *y* are offsets relative to the current graphics cursor position.

start and *end* are angles in **radians**. These can range $-2*\pi$ to $2*$ where π is the constant π , approximately 3.141593; if you specify a negative angle, its value is changed sign and a line is drawn from the up to that point in the arc. *end* angle can be less than *start*. If you do specify *start* and *end*, a full circle/ellipse is drawn; if you specify *s* but not *end*, *end* is assumed to be $2*\pi$; if you specify *end* but not *start* *start* is assumed to be 0.0.

aspect is the aspect ratio, or the ratio of the y radius over the x radius. omitted, the default for **ScreenRes** modes is 1.0, while for **Screen** mode default value is the value required to draw a perfect circle on the screen keeping the pixel aspect ratio in mind. This value can be calculated as follows:

$$ratio = (y_radius / x_radius) * pixel_aspect_ratio$$

Where *pixel_aspect_ratio* is the ratio of the current mode width over current mode height, assuming a 4:3 standard monitor. If aspect ratio than 1.0, radius is the x radius; if aspect is more or equal to 1.0, radius is the y radius.

F is the fill flag. If you specify this flag, the circle/ellipse will be filled with selected color. This only takes effect if you are drawing a full circle/ellipse.

Custom coordinates system set up by **Window** and/or **View (Graphics)** the drawing operation; clipping set by **View** also applies. When **circle** finishes drawing, the current graphics cursor position is set to the super center.

Example

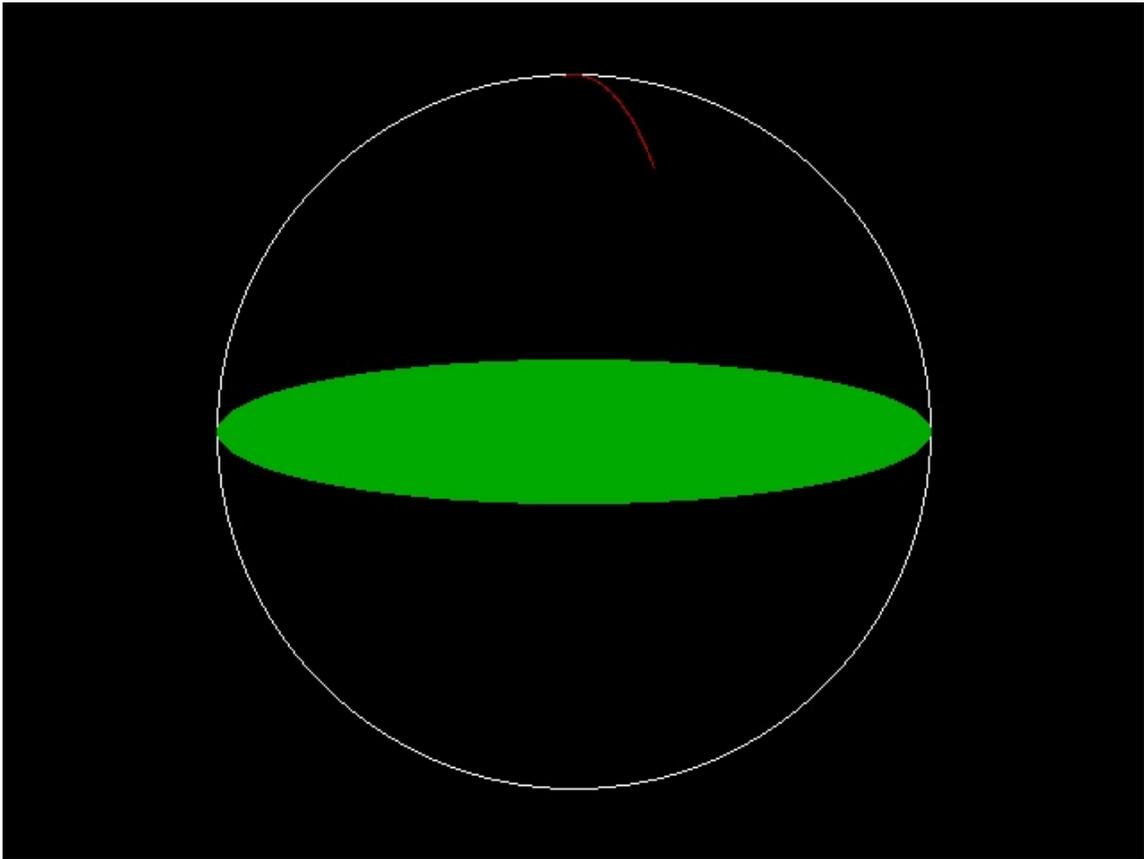
```
' Set 640x480 mode, 256 colors
Screen 18

' Draws a circle in the center
Circle (320, 240), 200, 15

' Draws a filled ellipse
Circle (320, 240), 200, 2, , , 0.2, F

' Draws a small arc
Circle (320, 240), 200, 4, 0.83, 1.67, 3

Sleep
```



Differences from QB

- *target* is new to FreeBASIC
- The FreeBASIC implementation uses a different algorithm for ellipse/arc drawing than QB, so the result may not be equal to QB for every pixel.
- The *F* flag to draw filled circles/ellipses is new to FreeBASIC.

See also

- [Screen \(Graphics\)](#)
- [Color](#)

Class



Declares a class object

Syntax

```
Class typename ...
```

Parameters

typename
name of the `class`

Description

We would have put something useful here (honest) except this feature isn't implemented in the compiler yet. But since it will get added in future, and there are several other document pages that need to link here, we thought it safe to include in anyway.

Example

```
' ' sample code
```

Output:

```
sample output
```

Dialect Differences

- Object-related features are supported only in the *-lang fb* optic

Differences from QB

- New to FreeBASIC

See also

- [Enum](#)
- [Type](#)

Clear



Clears or initializes some memory

Syntax

```
Declare Sub Clear cdecl ( ByRef dst As Any, ByVal value As Long
```

Usage

```
Clear( dst, [value], bytes )
```

Parameters

dst
starting address of some memory
value
the value to set all bytes equal to
bytes
number of bytes to clear

Description

`clear` sets one or more bytes in memory to a certain value (the default is 0). The starting address is taken from a reference to a variable or array element.

NOTE: In order to clear memory referenced by a **Pointer**, it must be cleared by the `clear` function. It will try to clear the bytes at the **pointer variable's** memory location.

Example

```
'create an array with 100 elements
Dim array(0 To 99) As Integer

'clear the contents of the array to 0, starting with 0
Clear array(0), , 100 * SizeOf(Integer)

'allocate 20 bytes of memory
Dim As Byte Ptr p = Allocate(20)
```

```
'set each of the first ten bytes to 0
Clear *p, 0, 10

'set each of the next ten bytes to 42
Clear p[10], 42, 10

'check the values of the allocated bytes
For i As Integer = 0 To 19
    Print i, p[i]
Next

'deallocate memory
Deallocate p
```

Differences from QB

- The behavior and usage is new to FreeBASIC
- The keyword `CLEAR` was used in QB to erase all variables, close the stack size. This use is not supported in FreeBASIC.

See also

- [Erase](#)
- [Reset](#)

Converts numeric or string expression to **Long**

Syntax

```
Declare Function CLng ( ByVal expression As datatype ) As Long
```

```
Type typename
```

```
Declare Operator Cast ( ) As Long
```

```
End Type
```

Usage

```
result = CLng( numeric expression )
```

```
result = CLng( string expression )
```

```
result = CLng( user defined type )
```

Parameters

expression

a numeric, string, or pointer expression to cast to a **Long** value

datatype

any numeric, string, or pointer data type

typename

a user defined type

Return Value

A **Long** value.

Description

The **CLng** function rounds off the decimal part and returns a 32-bit **Long**. The function does not check for an overflow, and results are undefined values which are less than -2 147 483 648 or larger than 2 147 483 648.

The name can be explained as 'Convert to LoNG'.

If the argument is a string expression, it is converted to numeric by using **ValInt**.

Example

```
' Using the CLNG function to convert a numeric val  
  
'Create an LONG variable  
Dim numeric_value As Long  
  
'Convert a numeric value  
numeric_value = CLng(-300.23)  
  
'Print the result, should return -300  
Print numeric_value  
Sleep
```

Differences from QB

- The string argument was not allowed in QB

See also

- CByte
- CByte
- CShort
- CUShort
- CInt
- CUInt
- CULng
- CLngInt
- CULngInt
- CSng
- CDbl

Converts numeric or string expression to 64-bit integer (**LongInt**)

Syntax

```
Declare Function CLngInt ( ByVal expression As datatype ) As Lon
```

```
Type typename
```

```
Declare Operator Cast ( ) As LongInt
```

```
End Type
```

Usage

```
result = CLngInt( numeric expression )
```

```
result = CLngInt( string expression )
```

```
result = CLngInt( user defined type )
```

Parameters

expression

a numeric, string, or pointer expression to cast to a **LongInt** value

datatype

any numeric, string, or pointer data type

typename

a user defined type

Return Value

A **LongInt** value.

Description

The **CLngInt** function rounds off the decimal part and returns a 64-bit integer. The function does not check for an overflow, and results are undefined for values which are less than -9 223 372 036 854 775 808 or larger than 9 223 372 036 854 775 807#.

The name can be explained as 'Convert to LoNG INTeger'.

If the argument is a string expression, it is converted to numeric. **ValLng**.

Example

```
' Using the CLNGINT function to convert a numeric  
  
'Create an LONG INTEGER variable  
Dim numeric_value As LongInt  
  
'Convert a numeric value  
numeric_value = CLngInt(-12345678.123)  
  
'Print the result, should return -12345678  
Print numeric_value  
Sleep
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the `__ClnInt`.

Differences from QB

- New to FreeBASIC

See also

- CByte
- CByte
- CShort
- CUShort
- CInt
- CUInt
- CLng
- CULng
- CULngInt

- CSng
- CDb1

Close



Stream I/O function to terminate access to a device

Syntax

```
Close [[#]filenum ] [, [#]filenum ...]  
or  
result = Close( [#filenum] )
```

Parameters

filenum
List of file numbers to close.

Description

Closes the files whose file numbers are passed as arguments. If an `close` without arguments closes all the files presently opened.

Terminating the program using an `End` statement will automatically close all files.

Return Value

`close` returns zero (0) on success and a non-zero error code otherwise.

Example

```
' Create a string and fill it.  
Dim buffer As String, f As Integer  
  
buffer = "Hello World within a file."  
  
' Find the first free file number.  
f = FreeFile  
  
' Open the file "file.ext" for binary usage, using  
Open "file.ext" For Binary As #f
```

```
' Place our string inside the file, using number
Put #f, , buffer

' Close the file. We could also do 'Close #f', bu
Close

' End of program. (Check the file "file.ext" upon
```

Differences from QB

- `close` can be called as a function that returns an error code.
- FB throws an error on trying to close an unused file number, if

See also

- `Open`
- `Put (File I/O)`
- `Get (File I/O)`
- `FreeFile`

Clears the screen in both text modes and graphics modes

Syntax

```
Declare Sub Cls ( ByVal mode As Long = 1 )
```

Usage

```
Cls mode
```

Parameters

mode

A optional numeric variable with a value from 0 to 2. If omitted, it default

Description

An optional *mode* parameter may be given,

If omitted, `cls` clears either the text or graphics viewport. If a graphics (**Graphics**) statement, the graphics viewport is cleared. Otherwise, the cleared. (If there is no explicit text viewport defined, the entire screen

If 0, clears the entire screen

If 1, clears the graphics viewport if defined. Otherwise, clears the text

If 2, clears the text viewport

Example

```
' set the color to light grey text on a blue back  
Color 7, 1  
  
' clear the screen to the background color  
Cls
```

```
' ' print text in the center of the screen
Locate 12, 33
Print "Hello Universe!"
```

In graphics modes, if you want to clear the entire screen to color 0, it costs more of the screen memory than calling `cls`.

```
Dim scrbuf As Byte Ptr, scrsz As Integer
Dim As Integer scrhei, scrpitch
Dim As Integer r = 0, dr = 1

ScreenRes 640, 480, 8

scrbuf = ScreenPtr: Assert( scrbuf <> 0 )
ScreenInfo( , scrhei, , , scrpitch )
scrsz = scrpitch * scrhei

Do

    ' ' lock the screen (must do this while working)
    ScreenLock

        ' ' clear the screen (could use cls here):
        Clear *scrbuf, 0, scrsz

        ' ' draw circle
        Circle (320, 240), r

    ScreenUnlock

    ' ' grow/shrink circle radius
    r += dr
    If r <= 0 Then dr = 1 Else If r >= 100 Then dr = -1

    ' ' short pause in each frame (prevents hogging)
```

```
Sleep 1, 1

'' run loop until user presses a key
Loop Until Len(Inkey) > 0
```

Differences from QB

- None

See also

- [Color](#)
- [Locate](#)
- [\(Print | ?\)](#)
- [View \(Graphics\)](#)

Color



Sets the display foreground / background color that is used with console graphics output of text

Syntax

```
Declare Function Color ( ByVal foreground As Long , ByVal backgr  
 ) As Long
```

Usage

```
Color [foreground] [, background]  
result = Color [( [foreground] [, background] )]
```

Parameters

foreground
the foreground color to set
background
the background color to set

Return Value

Returns a 32-bit value containing the current foreground color in the **L** the current background color in the **High Word**. (In hi/truecolor modes: foreground color is returned, taking up the whole 32 bits.)
The old color values can be retrieved at the same time as setting new

Description

The **color** statement sets the current foreground and/or background c
Draw, **Line (Graphics)**, **Cls**, **Paint**, **Print**, **PReset** and **PSet** all use the l
by this function when you don't specify a color to them, where applica
values that **color** accepts depend on the current graphics mode.

Mode	Meaning
1	foreground is screen color (ranging 0-15). background is the emulated CGA palette to (green, red, and brown), 1 (cyan, magenta and white), 2 (same as 0, but with bright c (same as 1, but with bright colors)
	foreground is a color index in current palette (ranging 0-1). background is a color inde

2, 11	palette (ranging 0-1).
7, 8	foreground is a color index in current palette (ranging 0-15). background is screen color in current palette (ranging 0-15).
9	foreground is a color index in current palette (ranging 0-63). background is screen color in current palette (ranging 0-63).
12	foreground is a color index in current palette (ranging 0-15). background is a color index in current palette (ranging 0-15).
13 and up	foreground is a color index in current palette (ranging 0-255). background is a color index in current palette (ranging 0-255).

If you are using a color depth higher than 8bpp, foreground and background are specified as direct **RGB** color values in the form `&h;AARRGGBB`, where *AA*, *RR*, *GG* and *BB* are the alpha, red, green and blue components ranging `&h00;-&hFF`; (0-255 in hexadecimal notation). While in hi/truecolor modes, you can use the **RGB** or **RGBA** macro to specify a valid color value.

A **Default Palette** is automatically set when entering a **Screen** mode.

Example

```
' Sets 320x240 in 32bpp color depth
Screen 14, 32

' Sets orange foreground and dark blue background
Color RGB(255, 128, 0), RGB(0, 0, 64)

' Clears the screen to the background color
Cls

' Prints "Hello World!" in the middle of the screen
Locate 15, 14
Print "Hello World!"

Sleep
```



```
Dim c As UInteger

'retrieve current color values
c = Color()

'extract color values from c using LOWORD and HIWORD
Print "Console colors:"
Print "Foreground: " & LoWord(c)
Print "Background: " & HiWord(c)
```

Differences from QB

- Direct color modes were not supported in QB.
- There is no border argument.

See also

- RGB
- RGBA
- LoWord
- HiWord
- Locate
- Palette

- **Screen**

Command



Returns command line parameters used to call the program

Syntax

```
Declare Function Command ( ByVal index As Long = -1 ) As String
```

Usage

```
result = Command[$]( [ index ] )
```

Parameters

index

Zero-based index for a particular command-line argument.

Return Value

Returns the command-line arguments(s).

Description

`Command` returns command-line arguments passed to the program upon

If *index* is less than zero (< 0), a space-separated list of all command-line arguments; a value of zero (0) returns the name of the executable; and values of one (1) and greater return individual arguments.

If *index* is greater than the number of arguments passed to the program, an empty string is returned.

When the command line is parsed for arguments, everything between double quotes is returned without the double quotes.

By default, filename globbing for arguments (expansion of wildcards to files) and redirection are typically not returned unless properly quoted. Consult the documentation for quoting of command line arguments.

WARNING: *By nature of constructor precedence in FreeBASIC and the order of constructor or UDT constructor called for static/shared object) is not s*

Disabling filename globbing under Win32

Define the following global variable(s) somewhere in the source:

```
' ' For MinGW.org and Cygwin runtimes:  
Extern _CRT_glob Alias "_CRT_glob" As Long  
Dim Shared _CRT_glob As Long = 0  
  
' ' For MinGW-w64 runtime:  
Extern _dowildcard Alias "_dowildcard" As Long  
Dim Shared _dowildcard As Long = 0
```

Disabling filename globbing under Dos

Define the following function somewhere in the source:

```
Function __crt0_glob_function Alias "__crt0_glob_f  
    Return 0  
End Function
```

Disabling filename globbing under Linux

Filename globbing is handled by the command shell. Quote the argun executing the command. For example in bash use 'set -f' to disable wi

Example

```
Print "program launched via: " & Command(0)  
  
Dim As Integer i = 1  
Do  
    Dim As String arg = Command(i)  
    If Len(arg) = 0 Then  
        Exit Do  
    End If  
  
    Print "command line argument " & i & " = "" &  
    i += 1
```

```
Loop

If i = 1 Then
    Print "(no command line arguments)"
End If

Sleep
```

Dialect Differences

- The string type suffix \$ is obligatory in the *-lang qb* dialect.
- The string type suffix \$ is optional in the *-lang fblite* and *-lang*

Differences from QB

- The numeric argument was not supported in QB.
- QB converted the parameter list to uppercase before returning
- By default arguments containing wildcard characters are expar

See also

- [__FB_ARGC__](#)
- [__FB_ARGV__](#)
- [Exec](#)
- [Run](#)

Variable declaration and scope modifier

Syntax

```
Common [Shared] symbolName[()] [AS DataType] [, ...]
```

Description

Declares a variable which is shared between code modules. A matching `Shared` must appear in all other code modules using the variable.

The `shared` optional parameter makes the variable global so that it can be used in `Subs` and `Functions`, as well as at module level. `common` arrays are always one-dimensional and must be defined with an empty parameter list `()`, and its dimensionality must be specified in a `ReDim` statement.

Example

```
' ' common1.bas  
  
Declare Sub initme()  
  
Common Shared foo() As Double  
  
ReDim foo(0 To 2) As Double  
  
initme()  
  
Print foo(0), foo(1), foo(2)
```

```
' ' common2.bas  
  
Common Shared foo() As Double
```

```
Sub initme()  
  foo(0) = 4*Atn(1)  
  foo(1) = foo(0)/3  
  foo(2) = foo(1)*2  
End Sub
```

Output:

3.141592653589793

1.047197551196598

2.094

Differences from QB

- The arrays will be always variable-length.
- *blockname* is not needed and must be removed because the original longer matters, only the symbol names.

See also

- **Dim**
- **Erase**
- **Extern**
- **LBound**
- **ReDim**
- **Preserve**
- **Shared**
- **Static**
- **UBound**
- **Var**

CondBroadcast



Restarts all threads `CondWaiting` for the handle

Syntax

```
Declare Sub CondBroadcast ( ByVal handle As Any Ptr )
```

Usage

```
CondBroadcast ( handle )
```

Parameters

handle

The handle of a conditional variable, or the null pointer (0) on failure.

Description

Once the conditional is `CondCreate` and the threads are started, one of more of them (including the main thread executing main program) can be set to `CondWait` for the conditional, they will be stopped until some other thread `CondSignals` that the waiting thread can restart.

`CondBroadcast` can be used to restart all threads waiting for the conditional. At the end of the program `CondDestroy` must be used to avoid leaking resources in the OS.

Condbroadcast must be used instead of `CondSignal` to restart all threads waiting on the conditional.

Example

See `CondCreate`

Platform Differences

- **Condbroadcast** is not available with the DOS version / target of FreeBASIC, because multithreading is not supported by DO kernel nor the used extender.
- In Linux the threads are always started in the order they are

created, this can't be assumed in Win32. It's an OS, not a FreeBASIC issue.

Dialect Differences

- Threading is not allowed in *-lang qb*

Differences from QB

- New to FreeBASIC

See also

- **CondCreate**
- **CondDestroy**
- **CondSignal**
- **CondWait**
- **ThreadCreate**

CondCreate



Creates a conditional variable to be used in synchronizing threads

Syntax

```
Declare Function CondCreate ( ) As Any Ptr
```

Usage

```
result = CondCreate
```

Return Value

A handle to a newly created conditional variable, or the null pointer (0)

Description

Once the conditional is **Condcreated** and the threads are started, one set to **CondWait** for the conditional, they will be stopped until some other thread is used to restart all threads waiting for the conditional. At the end of the

Example

See also [CondWait](#) and [CondSignal](#)

```
''  
'' make newly-created threads wait until all threads are ready  
''  
  
Dim Shared hcondstart As Any Ptr  
Dim Shared hmutexstart As Any Ptr  
Dim Shared start As Integer = 0  
  
Dim Shared threadcount As Integer  
Dim Shared hmutexready As Any Ptr  
Dim Shared hcondready As Any Ptr  
  
Sub mythread(ByVal id_ptr As Any Ptr)
```

```

Dim id As Integer = Cast(Integer, id_ptr)

'' signal that this thread is ready
MutexLock hmutexready
threadcount += 1
Print "Thread #" & id & " is waiting..."
CondSignal hcondready
MutexUnlock hmutexready

'' wait for the start signal
MutexLock hmutexstart
Do While start = 0
    CondWait hcondstart, hmutexstart
Loop

'' now this thread holds the lock on hmutexstart

MutexUnlock hmutexstart

'' print out the number of this thread
For i As Integer = 1 To 40
    Print id;
Next i
End Sub

Dim threads(1 To 9) As Any Ptr

hcondstart = CondCreate()
hmutexstart = MutexCreate()

hcondready = CondCreate()
hmutexready = MutexCreate()

threadcount = 0

MutexLock(hmutexready)
For i As Integer = 1 To 9
    threads(i) = ThreadCreate(@mythread, Cast(Any
    If threads(i) = 0 Then

```

```

        Print "unable to create thread"
    End If
Next i

Print "Waiting until all threads are ready..."

Do Until threadcount = 9
    CondWait(hcondready, hmutexready)
Loop
MutexUnlock(hmutexready)

Print
Print "Go!"

MutexLock hmutexstart
start = 1
CondBroadcast hcondstart
MutexUnlock hmutexstart

'' wait for all threads to complete
For i As Integer = 1 To 9
    If threads(i) <> 0 Then
        ThreadWait threads(i)
    End If
Next i

MutexDestroy hmutexready
CondDestroy hcondready

MutexDestroy hmutexstart
CondDestroy hcondstart

```

```

'Visual example of mutual exclusion + mutual synch
'by using Mutex and CondVar:
'the "user-defined thread" computes the points coc
'and the "main thread" plots the points.

```

```

'
'Principle of mutual exclusion + mutual synchronis
'          Thread#A          XOR + <==>
'
'.....
'MutexLock(mut)
'  While Thread#A_signal <> false
'    ConWait(cond, mut)
'  Wend
'  Do_something#A_with_exclusion
'  Thread#A_signal = true
'  ConSignal(cond)
'MutexUnlock(mut)
'.....
'
'Behavior:
'- Unnecessary to pre-calculate the first point.
'- Each calculated point is plotted one time only.
'
'If you comment out the lines containing "MutexLoc
'"ConWait" and "ConSignal", ".ready"
'(inside "user-defined thread" or/and "main thread
'there will be no longer mutual exclusion nor mutu
'between computation of coordinates and plotting c
'and many points will not be plotted on circle (du
'-----

Type ThreadUDT
  Dim handle As Any Ptr
  Dim sync As Any Ptr
  Dim cond As Any Ptr
  Dim ready As Byte
  Dim quit As Byte
  Declare Static Sub Thread (ByVal As Any Ptr)
  Dim procedure As Sub (ByVal As Any Ptr)
  Dim p As Any Ptr
  Const false As Byte = 0
  Const true As Byte = Not false
End Type

```

```

Static Sub ThreadUDT.Thread (ByVal param As Any Ptr
    Dim tp As ThreadUDT Ptr = param
    Do
        Static As Integer I
        MutexLock(tp->sync)
        While tp->ready <> false
            CondWait(tp->cond, tp->sync)
        Wend
        tp->procedure(tp->p)
        I += 1
        Locate 30, 38
        Print I;
        tp->ready = true
        CondSignal(tp->cond)
        MutexUnlock(tp->sync)
        Sleep 5
    Loop Until tp->quit = tp->true
End Sub

```

```

Type Point2D

```

```

    Dim x As Integer
    Dim y As Integer

```

```

End Type

```

```

Const x0 As Integer = 640 / 2
Const y0 As Integer = 480 / 2
Const r0 As Integer = 200
Const pi As Single = 4 * Atn(1)

```

```

Sub PointOnCircle (ByVal p As Any Ptr)

```

```

    Dim pp As Point2D Ptr = p
    Dim teta As Single = 2 * pi * Rnd
    pp->x = x0 + r0 * Cos(teta)
    Sleep 5

```

'To increas

```

    pp->y = y0 + r0 * Sin(teta)
End Sub

```

```

Screen 12
Locate 30, 2
Print "<any_key> : exit";
Locate 30, 27
Print "calculated:";
Locate 30, 54
Print "plotted:";

Dim Pptr As Point2D Ptr = New Point2D

Dim Tptr As ThreadUDT Ptr = New ThreadUDT
Tptr->sync = MutexCreate
Tptr->cond = CondCreate
Tptr->procedure = @PointOnCircle
Tptr->p = Pptr
Tptr->handle = ThreadCreate(@ThreadUDT.Thread, Tptr)

Do
    Static As Integer I
    Sleep 5
    MutexLock(Tptr->sync)           'Mutex (Loc
    While Tptr->ready <> Tptr->true  'Process lo
        CondWait(Tptr->cond, Tptr->sync) 'CondWait t
    Wend
    PSet (Pptr->x, Pptr->y)         'Plotting c
    I += 1
    Locate 30, 62
    Print I;
    Tptr->ready = Tptr->false       'Reset read
    CondSignal(Tptr->cond)         'CondSignal
    MutexUnlock(Tptr->sync)        'Mutex (Unl
Loop Until Inkey <> ""

MutexLock(Tptr->sync)           'Mutex (Loc
Tptr->ready = Tptr->false       'Reset read
Tptr->quit = Tptr->true         'Set quit
CondSignal(Tptr->cond)         'CondSignal

```

```
MutexUnlock(Tptr->sync)           'Mutex (Unl
ThreadWait(Tptr->handle)
MutexDestroy(Tptr->sync)
CondDestroy(Tptr->cond)
Delete Tptr
Delete Pptr

Sleep
```

See also the similar `MutexCreate` example

Platform Differences

- `Condcreate` is not available with the DOS version / target of Fi used extender.

Dialect Differences

- Threading is not allowed in *-lang qb*

Differences from QB

- New to FreeBASIC

See also

- `CondBroadcast`
- `CondDestroy`
- `CondSignal`
- `CondWait`
- `MutexCreate`
- `MutexLock`
- `MutexUnlock`
- `ThreadCreate`

CondDestroy



Destroys a multi-threading conditional variable when it is no more needed.

Syntax

```
Declare Sub CondDestroy ( ByVal handle As Any Ptr )
```

Usage

```
CondDestroy ( handle )
```

Parameters

handle

The handle of a conditional variable to destroy.

Description

Once the conditional is **CondCreated** and the threads are started, one or more of them (including the main thread executing main program) can be set to **CondWait** for the conditional, they will be stopped until some other thread **CondSignals** that the waiting thread can restart.

CondBroadcast can be used to restart all threads waiting for the conditional. At the end of the program **CondDestroy** must be used to avoid leaking resources in the OS.

Conddestroy destroys a condition variable, freeing the resources it might hold. No threads must be waiting on the condition variable on entrance to **Conddestroy**.

Example

See **CondCreate**, **CondWait** and **CondSignal**

Platform Differences

- **Conddestroy** is not available with the DOS version / target of FreeBASIC, because multithreading is not supported by DOS kernel nor the used extender.

Dialect Differences

- Threading is not allowed in *-lang qb*

Differences from QB

- New to FreeBASIC

See also

- **CondCreate**
- **CondBroadcast**
- **CondSignal**
- **CondWait**
- **ThreadCreate**

CondSignal



Restarts a thread suspended by a call to `CondWait`

Syntax

```
Declare Sub CondSignal ( ByVal handle As Any Ptr )
```

Usage

```
CondSignal ( handle )
```

Parameters

handle

The handle of a conditional variable, or the null pointer (0) on failure.

Description

Once the conditional is created with `CondCreate` and the threads are set to `CondWait` for the conditional, they will be stopped until some other thread calls `CondSignal` on all threads waiting for the conditional. At the end of the program `CondDestroy` must be called to destroy the conditional.

CondSignal restarts one thread waiting. It should be called after *mutex*. If the conditional is not waiting, nothing happens; if several are waiting, only one is restarted.

Example

See also `CondCreate` and `CondWait`

```
' This very simple example code demonstrates the use of CondSignal
' The main routine initializes a string and creates a thread
' The main routine waits until it receives the conditional signal
' The thread complements the string, then sends a signal
'
' Principle of mutual exclusion + simple synchronization
'          Thread#A          XOR + ==>
' .....
' MutexLock(mut)          Mut
```

```

' Do_something#A_with_exclusion                                W
' Thread#A_signal = true                                       W
' CondSignal(cond)                                            W
'MutexUnlock(mut)                                             C
' .....                                                       T
'                                                                Mut
'                                                                ...

Dim Shared As Any Ptr mutex
Dim Shared As Any Ptr cond
Dim Shared As String txt
Dim As Any Ptr pt
Dim Shared As Integer ok = 0

Sub thread (ByVal p As Any Ptr)
    Print "thread is complementing the string"
    MutexLock(mutex)
    Sleep 400
    txt &= " complemented by thread"
    ok = 1
    CondSignal(cond)
    MutexUnlock(mutex)
    Print "thread signals the processing completed"
End Sub

mutex = MutexCreate
cond = CondCreate

txt = "example of text"
Print "main() initializes a string = " & txt
Print "main creates one thread"
Print
pt = ThreadCreate(@thread)
MutexLock(mutex)
While ok <> 1
    CondWait(cond, mutex)
Wend
Print
Print "back in main(), the string = " & txt

```

```
ok = 0  
MutexUnlock(mutex)  
  
ThreadWait(pt)  
MutexDestroy(mutex)  
CondDestroy(cond)
```

Dialect Differences

- Threading is not allowed in *-lang qb*

Platform Differences

- **Condsignal** is not available with the DOS version / target of FreeBASIC
- In Linux the threads are always started in the order they are created

Differences from QB

- New to FreeBASIC

See also

- **CondCreate**
- **CondDestroy**
- **CondBroadcast**
- **CondWait**
- **ThreadCreate**

Stops execution of current thread until some condition becomes true

Syntax

```
Declare Sub CondWait ( ByVal handle As Any Ptr, ByVal mutex As A
```

Usage

```
CondWait ( handle, mutex )
```

Parameters

handle

The handle of a conditional variable, or the null pointer (0) on failure.

mutex

The mutex associated with this conditional variable, which must be locked

Description

Function that stops the thread where it is called until some other thread

Once the conditional variable is created with **condCreate** and the thread set to **condwait** for the conditional; they will be stopped until some other threads waiting for the conditional. At the end of the program **condDest**

When calling **condwait**, *mutex* should already be locked (using the same thread). When the condition variable will occur. The calling thread execution is suspended until the condition variable becomes signaled, *mutex* will be locked again after the thread is finished with it.

Note: It is a good habit to use **condwait** in a protected way against even if the condition variable is not signaled. For that, **condwait** is put within a loop for checking that a Boolean predicate is true when the thread has finished waiting.

See example below for detailed coding.

Example

See also **condCreate** and **condSignal**

```
' This simple example code demonstrates the use of  
' The main routine creates three threads.  
' Two of the threads update a "count" variable.  
' The third thread waits until the count variable
```

```
#define numThread 3  
#define countThreshold 6
```

```
Dim Shared As Integer count = 0  
Dim Shared As Any Ptr countMutex  
Dim Shared As Any Ptr countThresholdCV  
Dim As Any Ptr threadID(0 To numThread-1)  
Dim Shared As Integer ok = 0
```

```
Sub threadCount (ByVal p As Any Ptr)  
    Print "Starting threadCount(): thread#" & p  
    Do  
        Print "threadCount(): thread#" & p & ", loc  
        MutexLock(countMutex)  
        count += 1  
        ' Check the value of count and signal wait  
        ' Note that this occurs while mutex is loc  
        If count >= countThreshold Then  
            If count = countThreshold Then  
                Print "threadCount(): thread#" & p  
                ok = 1  
                ConSignal(countThresholdCV)  
            Else  
                Print "threadCount(): thread#" & p  
            End If  
            MutexUnlock(countMutex)  
            Exit Do  
        End If  
        Print "threadCount(): thread#" & p & ", cc  
        MutexUnlock(countMutex)  
        Sleep 100  
    Loop
```

```

End Sub

Sub threadWatch (ByVal p As Any Ptr)
    Print "Starting threadWatch(): thread#" & p &
    MutexLock(countMutex)
    ' Note that the Condwait routine will automati
    While ok = 0
        CondWait(countThresholdCV, countMutex)
    Wend
    Print "threadWatch(): thread#" & p & ", condit
    Print "threadWatch(): thread#" & p & ", count
    MutexUnlock(countMutex)
End Sub

' Create mutex and condition variable
countMutex = MutexCreate
countThresholdCV = CondCreate
' Create threads
threadID(0) = ThreadCreate(@threadWatch, Cast(Any
threadID(1) = ThreadCreate(@threadCount, Cast(Any
threadID(2) = ThreadCreate(@threadCount, Cast(Any
' Wait for all threads to complete
For I As Integer = 0 To numThread-1
    ThreadWait(threadID(I))
    Print "Main(): Waited on thread#" & I+1 & " Dc
Next I
MutexDestroy(countMutex)
CondDestroy(countThresholdCV)

```

Platform Differences

- **Condwait** is not available with the DOS version / target of Free
- In Linux the threads are always started in the order they are cr

Dialect Differences

- Threading is not allowed in *-lang qb*

Differences from QB

- New to FreeBASIC

See also

- **CondCreate**
- **CondDestroy**
- **CondBroadcast**
- **CondSignal**
- **MutexCreate**
- **MutexLock**
- **MutexUnlock**
- **ThreadCreate**

Const



Non-modifiable variable declaration.

Syntax

```
Const symbolname1 [AS DataType] = value1 [, symbolname2 [AS Data  
= value2, ...]  
or  
Const [AS DataType] symbolname1 = value1 [, symbolname2 = value2
```

Description

Declares non-modifiable constant data that can be integer or decimal (floating-point) numbers or strings. The constant type will be inferred if **DataType** isn't explicitly given.

Specifying **String * Size**, **Zstring * Size** Or **Wstring * Size** as **DataType** not allowed.

Specifying **String** as **DataType** is tolerated but without effect because the resulting type is always a **Zstring * Size**.

Example

```
Const Red = RGB(252, 2, 4)  
Const Black As UInteger = RGB(0, 0, 0)  
Const Text = "This is red text on a black bkgnd."  
  
Locate 1, 1  
Color Red, Black  
Print Text  
Sleep  
End
```

Differences from QB

- QB does not support the **As datatype** syntax.

See also

- `#define`
- `Const (Qualifier)`
- `Const (Member)`
- `Enum`
- `Var`

Const (Member)



Specifies that a member procedure is read only.

Syntax

```
Type typename
  Declare Const Sub|Function|Property|Operator ...
End Type

Const Sub|Function|... typename ...
...
End Sub|Function|...
```

Description

Specifies that a method does not change the object it is called on. The hidden **This** parameter will be considered read-only. The declaration can be read as 'invoking a const method promises not to change the object', and the compiler will error if the member procedure tries to change any of the data fields, or calls a non-const member procedure

Read-only (**const**) declarations are a measure of type safety that can be read as 'promises not to change.' The compiler uses the const declarations to check operations on variables and parameters and generate an error at compile time if their data could potentially change. There is no runtime overhead for using **const** qualifiers since all of the checks are made at compile time.

Constructors and destructors cannot be **const** (not useful). Member procedures can not be both **const** and **Static** since static member procedures do not have a hidden **This** parameter.

For methods with **const** in their declaration, **const** can also be specified on the corresponding method bodies, for improved code readability.

Example

```
' ' Const Member Procedures
```

```

Type foo
  x As Integer
  c As Const Integer = 0
  Declare Const Sub Inspect1()
  Declare Const Sub Inspect2()
  Declare Sub Mutate1()
  Declare Sub Mutate2()
End Type

''
Sub foo.Mutate1()
  '' we can change non-const data fields
  x = 1

  '' but we still can't change const data
  '' fields, they are promised not to change
  '' c = 1 '' Compile error

End Sub

''
Sub foo.Mutate2()
  '' we can call const members
  Inspect1()

  '' and non-const members
  Mutate1()

End Sub

''
Sub foo.Inspect1()
  '' can use data members
  Dim y As Integer
  y = c + x

  '' but not change them because Inspect1()
  '' is const and promises not to change foo
  '' x = 10 '' Compile error

```

```
End Sub

''
Sub foo.Inspect2()
    '' we can call const members
    Inspect1()

    '' but not non-const members
    '' Mutate1() '' Compile error

End Sub
```

Differences from QB

- New to FreeBASIC

See also

- **Const**
- **Const (Qualifier)**
- **Dim**
- **Type**

Const (Qualifier)



Specifies that a data type or pointer data type is read only.

Syntax

```
... As [Const] datatype [ [Const] Ptr ... ]
```

Parameters

datatype

Name of a standard or user defined data type.

Description

Specifies that the *datatype* or **Ptr** immediately to the right of the **const** qualifier is to be considered as read only. Read-only (**const**) declarations are a measure of type safety that can be read as 'promises not to change'. The compiler uses the **const** declarations to check operations on variables, parameters, and generate an error at compile time if their data could potentially change. There is no runtime overhead for using **const** qualifiers since all of the checks are made at compile time.

const can be used anywhere data type declarations are made. This includes variables, parameters, function return results, user defined type fields, aliases, and casting. The *datatype* can be any built-in standard data type or user defined type.

Read-only variables must have an initializer since modifying a read-only variable through an assignment will generate a compiler error. The initializer may appear after the declaration of the variable.

Both non-const and const variables may be passed to a procedure as a const parameter. However, a const variable may not be passed to a procedure taking a non-const parameter, and will generate a compile

Procedures can be overloaded based on the const-ness of parameter. For example, a procedure can be overloaded where one version of the procedure takes a 'byref foo as bar' parameter and another version of the pro

takes a 'byref foo as const bar' parameter.

With pointer declarations, **const** can be used to indicate which part of pointer declaration is read-only (all other parts are by default read-write). The read-only portion of the pointer data type could be the pointer itself (the address), what the pointer points to (the data), or both. In a declaration with more than one level of **Ptr** indirection, the right most **Ptr** indicates the order level of indirection and is therefore dereferenced first.

The compiler has an internal hard-limit of eight (8) levels of pointer indirection with respect to const qualifiers and the behavior of using **const** with **Ptr** types having greater than eight (8) levels of indirection is undefined.

Example

```
' ' Const Variables

' ' procedure taking a const parameter
Sub proc1( ByRef x As Const Integer )

    ' ' can't change x because it is const
    ' ' x = 10 ' ' compile error

    ' ' but we can use it in expressions and
    ' ' assign it to other variables
    Dim y As Integer
    y = x
    y = y * x + x

End Sub

' ' procedure taking a non-const parameter
Sub proc2( ByRef x As Integer )
    ' ' we can change the value
    x = 10
End Sub

' ' declare a non-const and const variable
```

```

Dim a As Integer
Dim b As Const Integer = 5

'' proc1() will accept a non-const or const
'' argument because proc1() promises not to
'' change the variable passed to it.
proc1( a )
proc1( b )

'' proc2() will accept a non-const argument
proc2( a )

'' but not a const argument because proc2()
'' might change the variable's data and we
'' promised that 'b' would not change.
'' proc2( b ) '' compile error

```

```

'' Const Pointers

'' an integer
Dim x As Integer = 1
Dim y As Integer = 2
Dim z As Integer = 3

'' To check that the compiler generates errors
'' when attempting to reassign const variables,
'' uncomment the assignments below.

''
Scope
'' a pointer to an integer
Dim p As Integer Ptr = @x

p = @y          '/' OK - pointer can be changed '/'
*p = z         '/' OK - data can be changed '/'

```

End Scope

''

Scope

'' a pointer to a constant integer

Dim p As Const Integer Ptr = @x

p = @y '/' OK - pointer can be changed '/'

'' *p = z '/' Error - data is const '/'

End Scope

''

Scope

'' a constant pointer to an integer

Dim p As Integer Const Ptr = @x

'' p = @y '/' Error - pointer is const '/'

*p = z '/' OK - data can be changed '/'

End Scope

''

Scope

'' a constant pointer to a constant integer

Dim p As Const Integer Const Ptr = @x

'' p = @y '/' Error - pointer is const '/'

'' *p = z '/' Error - data is const '/'

End Scope

'' Const Parameters in an Overloaded Procedure

'' procedure with non-const parameter

Sub foo Overload(ByRef n As Integer)

```

Print "called 'foo( byref n as integer )'"
End Sub

'' procedure with const parameter
Sub foo Overload( ByRef n As Const Integer )
    Print "called 'foo( byref n as const integer )'"
End Sub

Dim x As Integer = 1
Dim y As Const Integer = 2

foo( x )
foo( y )

'' OUTPUT:
'' called 'foo( byref n as integer )'
'' called 'foo( byref n as const integer )'

```

Differences from QB

- New to FreeBASIC

See also

- [Const](#)
- [Const \(Member\)](#)
- [Dim](#)
- [Type](#)

Constructor



Called automatically when a class or user defined type is created

Syntax

```
Type typename  
Declare Constructor ( )  
Declare Constructor ( [ ByRef | ByVal ] parameter As datatype [  
End Type  
  
Constructor typename ( [ parameters ] ) [ Export ]  
statements  
End Constructor
```

Parameters

typename
name of the **Type** or **Class**

Description

constructor methods are called when a user defined **Type** or **Class** va

typename is the name of the type for which the **constructor** method is
typename follows the same rules as procedures when used in a **Namesp**

More than one constructor may exist for a type or class. The exact co
signature matched when the variable is initialized. More than one *para*
declaration.

A constructor method is passed a hidden **This** parameter having the s
access the fields of the **Type** or **Class** which is to be initialized in the c

Constructors are called when declaring global or local static instances
dynamically using the **New** operator. See examples below for different

A copy **constructor** is a special constructor that initializes a new objec
cases where the copy **constructor** is called: when instantiating one of
instruction), when passing an object by value, when an object is return
statement).

Note: When an object is returned from a function by value, but by using assignment, the **constructor** is called once at first, and then the **Let** (A copy **constructor** must be defined if the shallow implicit copy constructor when the object manages dynamically allocated memory or other resource copied (for example if a member pointer points to dynamically allocated simply do an implicit pointer construction and a copy of value instead data).

Note: Even if is defined an explicit default **constructor**, it is never called

Chaining of constructors in nested types is supported. Any fields that The keyword **constructor**(*parameters*) can be used at the top of a constructor of same type. It prevents the compiler from emitting field initialization (to initialize everything).

constructor can be also called directly from the *typename* instance like same syntax, i.e. using a member access operator, e.g. *obj*.**Constructor**(*parameters*) *this*.**Constructor**(*parameters*) is not treated as chaining constructor, (constructors). In general it's not safe to manually call the constructor of the old object state - if any - is overwritten without any of its old member memory/resource leaks.

Example

Simple constructor example for beginners.

```
Type MyObj
  Foo As Integer Ptr

  ' Constructor to create our integer, and set
  Declare Constructor( ByVal DefVal As Integer = 0 )
  ' Destroy our integer on object deletion.
  Declare Destructor()
End Type

Constructor MyObj( ByVal DefVal As Integer = 0 )
  Print "Creating a new integer in MyObj!"
  Print "The Integer will have the value of: " & DefVal
  Print ""
```

```

    '' Create a pointer, and set its value to the
    '' Constructor.
    This.Foo = New Integer
    *This.Foo = DefVal
End Constructor

Destructor MyObj()
    Print "Deleting our Integer in MyObj!"
    Print ""

    '' Delete the pointer we created in MyObj.
    Delete This.Foo
    This.Foo = 0
End Destructor

Scope
    '' Create a MyObj type object
    '' Send the value of '10' to the constructor
    Dim As MyObj Bar = 10

    '' See if the integer's been created.  Print i
    Print "The Value of our integer is: " & *Bar.Foo
    Print ""

    Sleep
End Scope
    '' We've just gone out of a scope.  The Destruct
    '' Because our objects are being deleted.
Sleep

```

More advanced construction example, showing constructor overloading

```

Type sample
    _text As String

    Declare Constructor ()

```

```

Declare Constructor ( a As Integer )
Declare Constructor ( a As Single )
Declare Constructor ( a As String, b As Byte )

Declare Operator Cast () As String

End Type

Constructor sample ()
  Print "constructor sample ()"
  Print
  this._text = "Empty"
End Constructor

Constructor sample ( a As Integer )
  Print "constructor sample ( a as integer )"
  Print "  a = "; a
  Print
  this._text = Str(a)
End Constructor

Constructor sample ( a As Single )
  Print "constructor sample ( a as single )"
  Print "  a = "; a
  Print
  this._text = Str(a)
End Constructor

Constructor sample ( a As String, b As Byte )
  Print "constructor sample ( a as string, b as by
  Print "  a = "; a
  Print "  b = "; b
  Print
  this._text = Str(a) + "," + Str(b)
End Constructor

Operator sample.cast () As String
  Return this._text
End Operator

```

```

Print "Creating x1"
Dim x1 As sample

Print "Creating x2"
Dim x2 As sample = 1

Print "Creating x3"
Dim x3 As sample = 99.9

Print "Creating x4"
Dim x4 As sample = sample( "aaa", 1 )

Print "Values:"
Print "  x1 = "; x1
Print "  x2 = "; x2
Print "  x3 = "; x3
Print "  x4 = "; x4

```

Example of copy constructor.

```

Type UDT
  Dim As String Ptr p           ''pointer
  Declare Constructor ()       ''default
  Declare Constructor (ByRef rhs As UDT) ''copy constructor
  Declare Destructor ()       ''destructor
End Type

Constructor UDT ()
  This.p = CAllocate(1, SizeOf(String))
End Constructor

Constructor UDT (ByRef rhs As UDT)
  This.p = CAllocate(1, SizeOf(String))
  *This.p = *rhs.p
End Constructor

Destructor UDT ()

```

```
*This.p = ""
Deallocate This.p
End Destructor

Dim As UDT u0
*u0.p = "copy constructor exists"
Dim As UDT u = u0
*u0.p = "" ''to check the independance of the res
Print *u.p
Sleep
```

Dialect Differences

- Object-related features are supported only in the *-lang fb* option

Differences from QB

- New to FreeBASIC

See also

- [Class](#)
- [Constructor \(Module\)](#)
- [New](#)
- [Destructor](#)
- [Type](#)

Constructor (Module)



Specifies execution of a procedure before module-level code

Syntax

```
[Public | Private] Sub procedure_name [Alias "external_identifie  
[priority] [Static]  
{ procedure body }  
End Sub
```

Description

The **constructor** keyword is used in **Sub** definitions to force execution module-level code. Procedures defined as constructors may be used procedures, that is, they may be called from within module-level code

The procedure must have an empty parameter list. A compile-time error **constructor** keyword is used in a Sub definition having one or more p overloaded procedures, only one (1) constructor may be defined because multiple Subs which take no arguments.

In a single module, constructors normally execute in the reverse order

The *priority* attribute, an integer between 101 and 65535, can be used to execute in a certain order. The value of *priority* has no specific meaning when used with other constructor priorities. 101 is the highest priority and constructors having a *priority* attribute are executed before constructors with a *priority* value of 65535 is the same as not assigning a priority value.

A module may define multiple constructor procedures, and multiple modules may define constructors provided no two **Public** constructors share the same procedure name.

When linking with modules that also define constructors, the order of execution is determined at link-time unless the *priority* attribute is used. Therefore, special care must be taken with constructors that may call on a secondary module also defining a constructor. It is advisable to use a single constructor that explicitly calls initialization procedures.

Example

```

'' ConDesExample.bas : An example program that def
'' constructors and destructors. Demonstrates wher
'' they are called when linking a single module.

Sub Constructor1() Constructor
    Print "Constructor1() called"
End Sub

Sub Destructor1() Destructor
    Print "Destructor1() called"
End Sub

Sub Constructor2() Constructor
    Print "Constructor2() called"
End Sub

Sub Destructor2() Destructor
    Print "Destructor2() called"
End Sub

    '' -----
    Print "module-level code"

End 0
'' -----

```

Output:

```

Constructor2() called
Constructor1() called
module-level code
Destructor1() called
Destructor2() called

```

Differences from QB

- New to FreeBASIC

See also

- **Constructor (Class)**
- **Destructor (Module)**
- **Sub**

Continue



Control flow statement to continue next iteration of a loop

Syntax

```
Continue {Do | For | While}
```

Description

Skips all code until the end clause of a loop structure, i.e. **Do...Loop**, **For...Next**, **While...Wend** block, then executes the limit condition check. In the case of a **For...Next** block, the variable is incremented according to the **Step** specified.

Where there are multiple **Do / For / While** blocks nested, it will continue the innermost block of that type, i.e. the last one entered. You can continue an earlier block by giving the word multiple times, separated by commas. e.g. `continue w`

Example

```
Dim As Integer n

Print "Here are odd numbers between 0 and 10!"
Print
For n = 0 To 10

    If ( n Mod 2 ) = 0 Then
        Continue For
    End If

    Print n

Next n
```

```
' simple prime number finder
```

```

Print "Here are the prime numbers between 1 and 20"
Print

Dim n As Integer, d As Integer

For n = 2 To 20

    For d = 2 To Int(Sqr(n))

        If ( n Mod d ) = 0 Then ' d divides n
            Continue For, For ' n is not prime, so
        End If

    Next d

    Print n

Next n

```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- [Exit](#)

Cos



Returns the cosine of an angle

Syntax

```
Declare Function Cos ( ByVal angle As Double ) As Double
```

Usage

```
result = Cos( angle )
```

Parameters

angle
the angle (in radians)

Return Value

Returns the cosine of the argument *angle* as a **Double** within the range

Description

The argument *number* is measured in **radians** (not **degrees**).

The value returned by this function is undefined for values of *angle* with an absolute value of 2^{63} or greater.

Example

```
Const PI As Double = 3.1415926535897932
Dim a As Double
Dim r As Double
Input "Please enter an angle in degrees: ", a
r = a * PI / 180      'Convert the degrees to Radians
Print ""
Print "The cosine of a" ; a; " degree angle is"; Cos(r)
Sleep
```

Output:

```
Please enter an angle in degrees: 30  
The cosine of a 30 degree angle Is 0.8660254037844387
```

Differences from QB

- None

See also

- [Acos](#)
- [Sin](#)
- [Tan](#)
- [A Brief Introduction To Trigonometry](#)

Converts a pointer expression to a specified data type pointer

Syntax

```
CPtr( PointerDataType, expression )
```

Description

Converts *expression* to *PointerDataType*.

PointerDataType must be a **Pointer** type (e.g. a **DataType Ptr** or an **Array Ptr**). *expression* may have a different pointer type or be an **Integer**.

Note: Currently, FB does not actually enforce that PointerDataType matches expression's pointer type. Currently, it will display a warning if you try to convert with the `-lang qb` compiler switch.

Example

```
Dim intval As Integer
Dim intptr As Integer Ptr
intval = &h0080
intptr = @intval
' ' will print -128 and 128, as the first expression is an Integer
Print *CPtr( Byte Ptr, intptr ), *intptr
```

Dialect Differences

- Not available in the **-lang qb** dialect unless referenced with the `-lang qb` compiler switch.

Differences from QB

- New to FreeBASIC

See also

- **Ptr**
- **Cast**
- **CByte**
- **CShort**
- **CInt**
- **CLngInt**
- **CSng**
- **Cdbl**

Converts numeric or string expression to an integer (**short**)

Syntax

```
Declare Function CShort ( ByVal expression As datatype ) As Short
```

```
Type typename
```

```
Declare Operator Cast ( ) As Short
```

```
End Type
```

Usage

```
result = CShort( numeric expression )
```

```
result = CShort( string expression )
```

```
result = CShort( user defined type )
```

Parameters

expression

a numeric, string, or pointer expression to cast to a **short** value

datatype

any numeric, string, or pointer data type

typename

a user defined type

Return Value

A **short** value.

Description

The **cshort** function rounds off the decimal part and returns a 16-bit **short** value. The function does not check for an overflow, and results are undefined for values which are less than -32 768 or larger than 32 767.

The name can be explained as 'Convert to Short'.

If the argument is a string expression, it is converted to numeric by us

Example

```
' Using the CSHORT function to convert a numeric v  
  
'Create an SHORT variable  
Dim numeric_value As Short  
  
'Convert a numeric value  
numeric_value = CShort(-4500.66)  
  
'Print the result, should return -4501  
Print numeric_value  
Sleep
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the `__Cshort`.

Differences from QB

- New to FreeBASIC

See also

- `CByte`
- `CUByte`
- `CUShort`
- `CInt`
- `CUInt`
- `CLng`
- `CULng`
- `CLngInt`
- `CULngInt`

- CSng
- CDb1

Converts an expression to signed

Syntax

```
CSign ( expression )
```

Usage

```
variable = CSign ( expression )
```

Description

Converts an unsigned *expression* to a signed one, useful to force signed behavior of division or multiplication (including with [shl](#) and [shr](#)).

This is the opposite of [CUnsg](#).

Example

```
Dim value As UShort = 65535  
Print CSign(value) ' ' will print -1
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__Csign`.

Differences from QB

- New to FreeBASIC

See also

- [CUnsg](#)

Converts numeric or string expression to **single** precision floating point

Syntax

```
Declare Function CSng ( ByVal expression As datatype ) As Single
```

```
Type typename
```

```
Declare Operator Cast ( ) As Single
```

```
End Type
```

Usage

```
result = CSng( numeric expression )
```

```
result = CSng( string expression )
```

```
result = CSng( user defined type )
```

Parameters

expression

a numeric, string, or pointer expression to cast to a **single** value

datatype

any numeric, string, or pointer data type

typename

a user defined type

Return Value

A **single** precision value.

Description

The **csng** function returns a 32-bit **single** value. The function does not an overflow, so be sure not to pass a value outside the representable the **single** data type. The name can be explained as 'Convert to SiNG

If the argument to **csng** is a string expression, it is first converted to nu using **val**.

Example

```
' Using the CSNG function to convert a numeric val  
  
'Create an SINGLE variable  
Dim numeric_value As Single  
  
'Convert a numeric value  
numeric_value = CSng(-12345.123)  
  
'Print the result, should return -12345.123  
Print numeric_value  
Sleep
```

Differences from QB

- The string argument was not allowed in QB

See also

- CByte
- CByte
- CShort
- CUShort
- CInt
- CUInt
- CLng
- CULng
- CLngInt
- CULngInt
- CDb1

Returns the row position of the cursor

Syntax

```
Declare Function CsrLin ( ) As Integer
```

Usage

```
result = CsrLin
```

Return Value

An **Integer** specifying the current row of the cursor.

Description

Returns the current row the cursor is on (i.e. the "**cursor line**"). The topmost row is number 1.

Example

```
Print "The cursor is on row: "; CsrLin
```

Differences from QB

- None

See also

- **Locate**
- **Pos**

Converts numeric or string expression to an unsigned byte (**UByte**)

Syntax

```
Declare Function CUByte ( ByVal expression As datatype ) As UByte
```

```
Type typename
```

```
Declare Operator Cast ( ) As UByte
```

```
End Type
```

Usage

```
result = CUByte( numeric expression )
```

```
result = CUByte( string expression )
```

```
result = CUByte( user defined type )
```

Parameters

expression

a numeric, string, or pointer expression to cast to a **UByte** value

datatype

any numeric, string, or pointer data type

typename

a user defined type

Return Value

A **UByte** value.

Description

The **CUByte** function rounds off the decimal part and returns a 8-bit **UByte**. The function does not check for an overflow, and results are undefined which are less than 0 or larger than 255.

The name can be explained as 'Convert to Unsigned Byte'.

If the argument is a string expression, it is converted to numeric by us

Example

```
' Using the CUBYTE function to convert a numeric v  
  
'Create an UNSIGNED BYTE variable  
Dim numeric_value As UByte  
  
'Convert a numeric value  
numeric_value = CByte(123.55)  
  
'Print the result, should return 124  
Print numeric_value  
Sleep
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the `__cbyte`.

Differences from QB

- New to FreeBASIC

See also

- `CByte`
- `CShort`
- `CUShort`
- `CInt`
- `CUInt`
- `CLng`
- `CULng`
- `CLngInt`
- `CULngInt`

- CSng
- CDb1

Converts numeric or string expression to a **UInteger** or **UInteger<bits>**

Syntax

```
Declare Function CUInt ( ByVal expression As datatype ) As UInteger
Declare Function CUInt<bits> ( ByVal expression As datatype ) As UInteger<bits>
```

```
Type typename
Declare Operator Cast ( ) As UInteger
Declare Operator Cast ( ) As UInteger<bits>
End Type
```

Usage

```
result = CUInt( numeric expression )
result = CUInt( string expression )
result = CUInt( user defined type )
```

Parameters

bits

A numeric constant expression indicating the size in bits of unsigned integer desired. The values allowed are 8, 16, 32 or 64.

expression

a numeric, string, or pointer expression to cast to a **UInteger** or **UInteger** value

datatype

any numeric, string, or pointer data type

typename

a user defined type

Return Value

A **UInteger** or **UInteger<bits>** containing the converted value.

Description

The **CUInt** function rounds off the decimal part and returns a **UInteger** if a *bits* value is supplied, an unsigned integer type of the given size.

The function does not check for an overflow; for example, for a 32-bit results are undefined for values which are less than 0 or larger than 4

The name can be explained as 'Convert to Unsigned INTEger'.

If the argument is a string expression, it is converted to numeric by us or `ValULng`, depending on the size of the result type.

Example

```
' Using the CUINT function to convert a numeric va
'Create an UNSIGNED INTEGER variable
Dim numeric_value As UInteger
'Convert a numeric value
numeric_value = CUInt(300.23)
'Print the result = 300
Print numeric_value
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the `__Cuint`.

Differences from QB

- New to FreeBASIC

See also

- `CByte`
- `CByte`
- `CShort`

- **CUShort**
- **CInt**
- **CLng**
- **CULng**
- **CLngInt**
- **CULngInt**
- **CSng**
- **Cdbl**
- **UInteger**

Converts numeric or string expression to **Ulong**

Syntax

```
Declare Function CULng ( ByVal expression As datatype ) As Ulong
```

```
Type typename
```

```
Declare Operator Cast ( ) As Ulong
```

```
End Type
```

Usage

```
result = CULng( numeric expression )
```

```
result = CULng( string expression )
```

```
result = CULng( user defined type )
```

Parameters

expression

a numeric, string, or pointer expression to cast to a **Ulong** value

datatype

any numeric, string, or pointer data type

typename

a user defined type

Return Value

A **Ulong** value.

Description

The **cuLng** function rounds off the decimal part and returns a 32 bit **Ulong**. The function does not check for an overflow. The name can be explained as 'Convert to Unsigned LoNG'.

If the argument is a string expression, it is converted to numeric by using **ValUlong**.

Example

```
' Using the CULNG function to convert a numeric va  
  
'Create an UNSIGNED LONG variable  
Dim numeric_value As ULONG  
  
'Convert a numeric value  
numeric_value = CULng(300.23)  
  
'Print the result = 300  
Print numeric_value  
Sleep
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the `__CULng`.

Differences from QB

- New to FreeBASIC

See also

- CByte
- CByte
- CShort
- CUShort
- CInt
- CUInt
- CLng
- CLngInt
- CULngInt
- CSng
- CDb1

Converts numeric or string expression to 64-bit unsigned integer (**ULongInt**)

Syntax

```
Declare Function CULngInt ( ByVal expression As datatype ) As ULongInt
```

```
Type typename
```

```
Declare Operator Cast ( ) As ULongInt
```

```
End Type
```

Usage

```
result = CULngInt( numeric expression )
```

```
result = CULngInt( string expression )
```

```
result = CULngInt( user defined type )
```

Parameters

expression

a numeric, string, or pointer expression to cast to a **ULongInt** value

datatype

any numeric, string, or pointer data type

typename

a user defined type

Return Value

A **ULongInt** value.

Description

The **CULngInt** function rounds off the decimal part and returns a 64-bit value. The function does not check for an overflow, and results are unvalues which are less than 0 or larger than 18 446 744 073 709 551 615. casts from floating-point expressions are currently not guaranteed to higher than 2^{63} (9 223 372 036 854 775 808).

The name can be explained as 'Convert to Unsigned LoNG INTEger'.

If the argument is a string expression, it is converted to numeric by us

Example

```
' Using the CLNGINT function to convert a numeric
'Create an UNSIGNED LONG INTEGER variable
Dim numeric_value As ULongInt

'Convert a numeric value
numeric_value = CULngInt(12345678.123)

'Print the result, should return 12345678
Print numeric_value
Sleep
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the `__CuLngInt`.

Differences from QB

- New to FreeBASIC

See also

- `CByte`
- `CByte`
- `CShort`
- `CUShort`
- `CInt`
- `CUInt`
- `CLng`

- **CULng**
- **CLngInt**
- **CSng**
- **Cdbl**

Converts an expression to unsigned

Syntax

`CUnsg (expression)`

Usage

`variable = CUnsg (expression)`

Converts a signed *expression* to an unsigned one, useful to force unsigned behavior of division or multiplication (including with `shl` and `shr`).

This is the opposite of `csign`.

Example

```
Dim value As Short = -1
Print CUnsg(value) ' will print 65535
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__cunsg`.

Differences from QB

- New to FreeBASIC

See also

- `CSign`

CurDir



Returns the current directory/folder

Syntax

```
Declare Function CurDir ( ) As String
```

Usage

```
result = CurDir
```

Return Value

A **String** which is set to the name of the current directory/folder.

Description

Returns the current directory/folder.

Example

```
Print CurDir
```

output will vary.

Dialect Differences

- Not available in the **-lang qb** dialect unless referenced with the alias `__curdir`.

Differences from QB

- New to FreeBASIC

See also

- [Open](#)

- `Dir`
- `MkDir`
- `Rmdir`

Converts numeric or string expression to an unsigned integer (**ushort**)

Syntax

```
Declare Function CUShort ( ByVal expression As datatype ) As USh
```

```
Type typename
```

```
Declare Operator Cast ( ) As UShort
```

```
End Type
```

Usage

```
result = CUShort( numeric expression )
```

```
result = CUShort( string expression )
```

```
result = CUShort( user defined type )
```

Parameters

expression

a numeric, string, or pointer expression to cast to a **ushort** value

datatype

any numeric, string, or pointer data type

typename

a user defined type

Return Value

A **ushort** value.

Description

The **cushort** function rounds off the decimal part and returns a 16-bit **ushort**. The function does not check for an overflow, and results are undefined for values which are less than 0 or larger than 65 535.

The name can be explained as 'Convert to Unsigned Short'.

If the argument is a string expression, it is converted to numeric by us

Example

```
' Using the CUSHORT function to convert a numeric  
  
'Create an USHORT variable  
Dim numeric_value As UShort  
  
'Convert a numeric value  
numeric_value = CUShort(36000.4)  
  
'Print the result, should return 36000  
Print numeric_value  
Sleep
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the `__Cushort`.

Differences from QB

- New to FreeBASIC

See also

- CByte
- CByte
- CShort
- CInt
- CUInt
- CLng
- CULng
- CLngInt
- CULngInt
- CSng

- CDb1

Parameter to the **Put** graphics statement which selects a custom method

Syntax

```
Put [ target, ] [ STEP ] ( x,y ), source [ ,( x1,y1 )-( x2,y2 )
```

Parameters

Custom

Required.

custom_function_ptr

name of the custom user defined function.

parameter

optional **Pointer** to be passed to the custom function; if omitted, the d

Description

custom selects a custom user defined function as the method for blitting

The **custom** method uses a user-defined function to calculate the final source image, and will receive the source and destination pixel values the destination buffer. The function has the form:

```
Declare Function identifier ( _  
ByVal source_pixel As UInteger, _  
ByVal destination_pixel As UInteger, _  
ByVal parameter As Any Ptr _  
) As UInteger
```

identifier is the name of the function.

source_pixel is the current pixel value of the source image.

destination_pixel is the current pixel value of the destination image.

parameter is the parameter that is passed by the **Put** command. If it was

Example

```
Function dither ( ByVal source_pixel As UInteger, B
```

```

    'either returns the source pixel or the destin

Dim threshold As Single = 0.5
If parameter <> 0 Then threshold = *CPtr(Single

If Rnd() < threshold Then
    Return source_pixel
Else
    Return destination_pixel
End If

End Function

Dim img As Any Ptr, threshold As Single

' set up a screen
ScreenRes 320, 200, 16, 2
ScreenSet 0, 1

' create an image
img = ImageCreate(32, 32)
Line img, ( 0, 0)-(15, 15), RGB(255, 0, 0), b
Line img, (16, 0)-(31, 15), RGB( 0, 0, 255), b
Line img, ( 0, 16)-(15, 31), RGB( 0, 255, 0), b
Line img, (16, 16)-(31, 31), RGB(255, 0, 255), b

' dither the image with varying thresholds
Do Until Len(Inkey)

    Cls

    threshold = 0.2
    Put ( 80 - 16, 100 - 16), img, Custom, @dither,

    ' default threshold = 0.5
    Put (160 - 16, 100 - 16), img, Custom, @dither

```

```
threshold = 0.8
Put (240 - 16, 100 - 16), img, Custom, @dither,

ScreenCopy
Sleep 25

Loop

'' free the image memory
ImageDestroy img
```

Dialect Differences

- Not available in the *-lang qb* dialect.

Differences from QB

- New to FreeBASIC

See also

- [Put \(Graphics\)](#)

Converts a 64-bit integer or 8-byte string to a double-precision value

Syntax

```
Declare Function CVD ( ByVal l As LongInt ) As Double  
Declare Function CVD ( ByRef str As Const String ) As Double
```

Usage

```
result = CVD( l )  
result = CVD( str )
```

Parameters

l

A 64-bit **LongInt** with a binary copy of a double-precision variable stored in it.

str

A **String** at least 8 bytes in length with a binary copy of a double-precision variable stored in it.

Return Value

Returns a **Double** value holding a binary copy of the input value.

Description

Does a binary copy from a 64-bit **LongInt** or 8-byte **String** to a **Double** variable. A value of zero (0.0) is returned if the string is less than 8 bytes in length. The result will make sense only if the parameter contained a IEEE-754 formatted double-precision value, such as one generated by **CVLongInt** or **MKD**.

This function is useful to read numeric values from buffers without using a **Type** definition.

Example

```
Dim d As Double, l As LongInt
d = 1.125
l = CVLongInt(d)

Print Using "l = _&H&"; Hex(l)
Print Using "cvd(i) = &"; CVD(l)
```

Differences from QB

- QB did not support integer arguments.

See also

- MKD
- CVS
- CVLongInt

Converts a single-precision floating-point number or string to an integer variable using a binary copy

Syntax

```
Declare Function CVI ( ByVal sng As Single ) As Integer
Declare Function CVI ( ByRef str As Const String ) As Integer
Declare Function CVI<bits> ( expr As DataType ) As Integer<bits>
```

Usage

```
result = CVI( sng )
result = CVI( str )
result = CVI<bits>( expr )
```

Parameters

sng

A **Single** floating-point number with a binary copy of an integer variable stored in it.

str

A **String** with a binary copy of an integer variable stored in it.

bits

Specifies a size of integer type to return. The types and sizes of *expr* accepted will depend on the corresponding function called.

expr

An expression that will be copied into an **Integer**<*bits*>.

Return Value

An **Integer** or **Integer**<*bits*> variable containing a binary copy of the input expression.

Description

Returns an integer value using the binary data contained in a **Single**, or a **String**. A value of zero (0) is returned if the string contains fewer characters than the size of the return type.

`CVI` is used to convert strings created with `MKI`.

This function can also be used to convert 32-bit integer values from a memory or file buffer without the need for a `Type` structure. However, just as with the type structure, special care should be taken when using `CVI` to convert strings that have been read from a buffer.

`CVI` supports an optional `<bits>` parameter before the argument. If `bits` is 16, `CVShort` will be called instead; if `bits` is 32, `CVL` will be called; if `bits` is 64, `CVLongInt` will be called. The return type and accepted argument types will depend on which function is called. See each function's page for more information.

Example

```
Dim i As Integer, s As String
s = "ABCD"
i = CVI(s)
Print Using "s = "&""; s
Print Using "i = _&H&"; Hex(i)
```

Dialect Differences

- In the *-lang qb* dialect, `CVI` expects a 2-byte string, since a QB integer is only 16 bits. Only the first two bytes of the string are used, even if the string happens to be longer than two bytes.
- In the *-lang qb* dialect, `CVI` will not take a floating-point argument, since a QB integer is only 16 bits and there is no 16 bit floating-point data type. Instead, `CVI<32>/CVI<64>` or `CVL/CVLongInt` may be used.

Differences from QB

- In QB an error occurs if the string passed is fewer than two bytes in length.
- QB did not support floating-point arguments.

- QB did not support a *<bits>* parameter.

See also

- **MKI**
- **CVShort**
- **CVL**
- **CVLongInt**
- **Integer**

Converts a single-precision floating-point number or four-byte string to a integer (**Long**) variable

Syntax

```
Declare Function CVL ( ByVal sng As Single ) As Long
Declare Function CVL ( ByRef str As Const String ) As Long
```

Usage

```
result = CVL( sng )
result = CVL( str )
```

Parameters

sng

A **Single** floating-point number with a binary copy of an integer variable stored in it.

str

A **String** at least four bytes in length with a binary copy of an integer variable stored in it.

Return Value

A **Long** variable to copy the binary copy of a integer to.

Description

Returns a 32-bit **Long** integer value using the binary data contained in **Single**, or a **String** of at least four bytes in length. A value of zero (0) is returned if the string is less than four bytes in length.

CVL is used to convert 4-byte strings created with **MKL**.

This function can also be used to convert 32-bit integer values from a memory or file buffer without the need for a **Type** structure. However, just as with the type structure, special care should be taken when using **CVL** to convert strings that have been read from a buffer.

Example

```
Dim l As Long, s As String
s = "ABCD"
l = CVL(s)
Print Using "s = "&""; s
Print Using "l = &"; l
```

Differences from QB

- In QB an error occurs if the string passed is less than four byte in length.
- QB did not support floating-point arguments.

See also

- MKL
- CVShort
- CVI
- CVLongInt

Converts a double-precision floating-point number or eight-byte string to a **LongInt** variable

Syntax

```
Declare Function CVLongInt ( ByVal dbl As Double ) As LongInt
Declare Function CVLongInt ( ByRef str As Const String ) As
LongInt
```

Usage

```
result = CVLongInt( dbl )
result = CVLongInt( str )
```

Parameters

dbl

A **Double** floating-point number with a binary copy of a **LongInt** variable stored in it.

str

A **String** at least eight bytes in length with a binary copy of a **LongInt** variable stored in it.

Return Value

A **LongInt** variable holding a binary copy of the input variable.

Description

Returns a 64-bit **LongInt** value using the binary data contained in a **Double**, or a **String** of at least eight bytes in length. A value of zero (0) is returned if the string is less than eight bytes in length.

CVLongInt is used to convert 8-byte strings created with **MKLongInt**.

This function can also be used to convert 64-bit integer values from a memory or file buffer without the need for a **Type** structure. However, just as with the type structure, special care should be taken when using **CVLongInt** to convert strings that have been read from a buffer.

Example

```
Dim ll As LongInt, s As String
s = "ABCDEFGH"
ll = CVLongInt(ll)
Print Using "s = "&""; s
Print Using "ll = _&H&"; Hex(ll)
```

Differences from QB

- In QB an error occurs if the string passed is less than eight bytes in length.
- QB did not support floating-point arguments.

See also

- MKLongInt
- CVShort
- CVI
- CVL

Converts a 32-bit integer or 4-byte string to a single-precision variable

Syntax

```
Declare Function CVS ( ByVal i As Integer ) As Single  
Declare Function CVS ( ByRef str As Const String ) As Single
```

Usage

```
result = CVS( i )  
result = CVS( str )
```

Parameters

i

A 32-bit **Integer** with a binary copy of a single-precision variable stored in it.

str

A **String** at least 4 bytes in length with a binary copy of a single-precision variable stored in it.

Return Value

Returns a **single** value holding a binary copy of the input value.

Description

Does a binary copy from a 32-bit **Integer** or 4-byte **String** to a **Single** variable. A value of zero (0.0) is returned if the string is less than 4 bytes in length. The result will make sense only if the parameter contained a IEEE-754 formatted single-precision value, such as one generated by **CVI** or **MKS**.

This function is useful to read numeric values from buffers without using a **Type** definition.

Example

```
Dim f As Single, i As Integer
f = 1.125
i = CVI(f)

Print Using "i = _&H&"; Hex(i)
Print Using "cvs(i) = &"; CVS(i)
```

Differences from QB

- QB did not support integer arguments.

See also

- MKS
- CVD
- CVI

Converts a two-byte string to a **short** integer variable

Syntax

```
Declare Function CVShort ( ByRef str As Const String ) As Short
```

Usage

```
result = CVShort( str )
```

Parameters

str

A **String** at least two bytes in length with a binary copy of a **short** integer variable stored in it.

Return Value

short variable holding the binary copy of a **Keypgshort**.

Description

Returns a 16-bit **short** integer value using the binary data contained in a **String** of at least two bytes in length. A value of zero (0) is returned if the string is less than two bytes in length.

cvshort is used to convert 2-byte strings created with **MKShort**.

This function can also be used to convert 16-bit integer values from a memory or file buffer without the need for a **Type** structure. However, just as with the type structure, special care should be taken when using **cvshort** to convert strings that have been read from a buffer.

Example

```
Dim si As Short, s As String  
s = "AB"
```

```
si = CVShort(s)
Print Using "s = "&""; s
Print Using "si = _&H&"; Hex(si)
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__cvshort`.

Differences from QB

- In QB this function is called CVI

See also

- `MKShort`
- `CVI`
- `CVL`
- `CVLongInt`

Statement to store data at compile time.

Syntax

```
Data constant_expression1 [, constant_expression2]...
```

Description

Data stores a list of constant numeric or alphabetical expressions that compile time (except with *-lang qb*) and stored as constants that can be read by using **Read**.

All the **Data** statements in the program behave as a single chained list. When one **Data** statement is read, the first element of the following **Data** statement is read. The program should not attempt to **Read** after the last **Data** element. This is undefined in some dialects, and the program may crash (Page Fault).

Data statements are only visible from within the module in which they are entered. They can only be entered in module-level code.

Data constants can only be of simple types (numeric or string). A numeric constant is read into a string. A string is read into a numeric variable using the **Val** function. **Consts** can be used as items of data except in the *-lang qb* dialect. Variable names are considered as normal text.

The "**Restore label**" statement makes the first **Data** item after the *label* read, allowing the user to choose specific sections of data to read.

Data is normally used to initialize variables. FreeBASIC also allows the initialization of variables when they are **dimensioned** - see **Variable Initializers** for more information.

Example

```
' Create an array of 5 integers and a string to hold the data
Dim As Integer h(4)
Dim As String hs
```

```

Dim As Integer readindex

' Set up to loop 5 times (for 5 numbers... check t
For readindex = 0 To 4

    ' Read in an integer.
    Read h(readindex)

    ' Display it.
    Print "Number" ; readindex ; " = " ; h(readindex)

Next readindex

' Spacer.
Print

' Read in a string.
Read hs

' Print it.
Print "String = " + hs

' Await a keypress.
Sleep

' Exit program.
End

' Block of data.
Data 3, 234, 435/4, 23+433, 87643, "Good" + "Bye!"

```

Dialect Differences

- **-lang fb** and **-lang fbLite** considers data items as constant expressions evaluated during compilation and its result stored in the program.
- **-lang qb** considers unquoted words, including names of variables and literal strings, and stores them without change, as in QBASIC.

delimited by commas, and a colon or a line-break signifies the statement. Unquoted strings are trimmed of whitespace at the

Differences from QB

- Outside of the *-lang qb* dialect, alphabetic string literals must be enclosed in quotation marks, in QBASIC this was optional.
- In QBASIC empty items evaluated to number 0 or to empty string give a compile error. In QBASIC a comma at the end of the statement is an additional, empty item, evaluated to 0 or an empty string. In FreeBASIC a comma at the end of the statement gives a compile error.

See also

- [Read](#)
- [Restore](#)

Date



Returns the current system date as a string

Syntax

```
Declare Function Date ( ) As String
```

Usage

```
result = Date
```

Return Value

Returns the current system date, in the format mm-dd-yyyy

Description

None

Example

```
Print Date ' prints the current date
```

Differences from QB

- The QB DATE statement (to set the system date) is now called `SetDate`.

See also

- `SetDate`
- `Time`
- `Timer`

DateAdd



Offset a date with a specified interval

Syntax

```
Declare Function DateAdd ( ByRef interval As Const String, ByVal  
number As Double, ByVal date_serial As Double ) As Double
```

Usage

```
#include "vbcompat.bi"  
result = DateAdd( interval, number, date_serial )
```

Parameters

interval

string indicating which period of time corresponds to one unit of *number*

the number of intervals to add to the base date. The number will be rounded to the nearest integer.

date_serial

the base date

Return Value

Returns a **Date Serial** corresponding to the received *date_serial* plus the *number* of *intervals*.

Description

Interval is specified as follows:

value	interval
yyyy	years
q	quarter(three months)
m	months
ww	weeks
d,w,y	days
h	hours

n	minutes
s	seconds

The compiler will not recognize this function unless `vbcompat.bi` or `datetime.bi` is included.

Example

```
#include "vbcompat.bi"

Const fmt = "dddddd ttttt"
Dim d As Double
d = Now()

Print "1 hour from now is ";
Print Format( DateAdd( "h", 1, d ), fmt )

Print "1 day from now is ";
Print Format( DateAdd( "d", 1, d ), fmt )

Print "1 week from now is ";
Print Format( DateAdd( "ww", 1, d ), fmt )

Print "1 month from now is ";
Print Format( DateAdd( "m", 1, d ), fmt )
```

Differences from QB

- Did not exist in QB. This function appeared in Visual Basic.

See also

- **Date Serials**

DateDiff



Gets the difference of two dates measured by a specified interval

Syntax

```
Declare Function DateDiff ( ByRef interval As Const String, ByVa  
ByVal serial1 As Double, ByVal firstdayofweek As Long = fbUseSys  
firstdayofyear As Long = fbUseSystem ) As Long
```

Usage

```
#include "vbcompat.bi"  
result = DateDiff( interval, date_serial1, date_serial2 [, first  
firstweekofyear ] ] )
```

Parameters

interval
the unit of time (interval) with which to measure the difference

date_serial1
starting date serial

date_serial2
end date serial

firstdayofweek
first day of the week

firstdayofyear
first day of the year

Return Value

Returns an integer corresponding to the number of *intervals* found b

If *date_serial1* > *date_serial2*, the result is negative.

Description

interval is specified as follows:

value	interval

yyyy	years
q	quarter(three months)
m	months
w	seven day periods
ww	calendar weeks
d,y	days
h	hours
n	minutes
s	seconds

first_dayofweek Affects the counting when 'ww' interval is used.

value	first day of week	constant
omitted	sunday	
0	local settings	fbUseSystem
1	sunday	fbSunday
2	monday	fbMonday
3	tuesday	fbTuesday
4	wednesday	fbWednesday
5	thursday	fbThursday
6	friday	fbFriday
7	saturday	fbSaturday

first_weekofyear specifies which year (previous or next) that the week one year and the beginning of the next should included with.

value	first week of year	constant
0	local settings	fbUseSystem
1	January 1's week	fbFirstJan1
2	first weeks having 4 days in the year	fbFirstFourDays
3	first full week of year	fbFirstFullWeek

Notice if you do an arithmetical subtraction of two date serials you get

The compiler will not recognize this function unless `vbcompat.bi` OR `dat`

Example

```
#include "vbcompat.bi"

Dim s As String, d1 As Double, d2 As Double

Line Input "Enter your birthday: ", s

If IsDate( s ) Then
    d1 = DateValue( s )
    d2 = Now()

    Print "You are " & DateDiff( "yyyy", d1, d2 ) & " years old"
    Print "You are " & DateDiff( "d", d1, d2 ) & " days old"
    Print "You are " & DateDiff( "s", d1, d2 ) & " seconds old"

Else
    Print "Invalid date"

End If
```

Differences from QB

- Did not exist in QB. This function appeared in Visual Basic.

See also

- **Date Serials**

DatePart



Gets an interval from a date

Syntax

```
Declare Function DatePart ( ByRef interval As Const String, ByVa  
date_serial As Double, ByVal firstdayofweek As Long =  
fbUseSystem, ByVal firstdayofyear As Long = fbUseSystem ) As Lon
```

Usage

```
#include "vbcompat.bi"  
result = DatePart( interval, date_serial, first_dayofWeek [,  
first_week_of_year ] )
```

Parameters

interval
string indicating which part of the date is required
date_serial
the date serial to decode
firstdayofweek
first day of the week
firstdayofyear
first day of the year

Return Value

Return an integer representing the *interval* in the **Date Serial**.

Description

interval string indicating which part of the date is required is specified as follows:

value	interval
yyyy	years
q	quarter(three months)
m	months

w	weekday
ww	week of the year
y	day of the year
d	day of the month
h	hours
n	minutes
s	seconds

first_dayofweek Affects the output when 'w' interval is required.

value	first day of week	constant
omitted	sunday	
0	local settings	fbUseSystem
1	sunday	fbSunday
2	monday	fbMonday
3	tuesday	fbTuesday
4	wednesday	fbWednesday
5	thursday	fbThursday
6	friday	fbFriday
7	saturday	fbSaturday

first_weekofyear specifies which year (previous or next) that the week which spans the end of one year and the beginning of the next should included with. Affects the output when 'ww' interval is required.

value	first week of year	constant
0	local settings	fbUseSystem
1	January 1's week	fbFirstJan1
2	first weeks having 4 days in the year	fbFirstFourDays
3	first full week of year	fbFirstFullWeek

The compiler will not recognize this function unless `vbcompat.bi` or `datetime.bi` is included.

Example

```
#include "vbcompat.bi"

Dim d As Double

d = Now()

Print "Today is day " & DatePart( "y", d );
Print " in week " & DatePart( "ww", d );
Print " of the year " & DatePart( "yyyy", d )
```

Differences from QB

- Did not exist in QB. This function appeared in Visual Basic.

See also

- **Date Serials**

DateSerial



Creates a **date serial**

Syntax

```
Declare Function DateSerial ( ByVal year As Long, ByVal month As Long, ByVal day As Long ) As Long
```

Usage

```
#include "vbcompat.bi"  
result = DateSerial( year, month, day )
```

Parameters

year
the year
month
the month of the year
day
the day of the month

Return Value

Returns a **date serial** containing the date formed by the values in the *year*, *month* and *day* parameters. The date serial returned has no decimal part.

Description

The compiler will not recognize this function unless `vbcompat.bi` or `datetime.bi` is included.

Example

```
#include "vbcompat.bi"  
  
Dim a As Double = DateSerial(2005, 11, 28)
```

```
Print Format(a, "yyyy/mm/dd hh:mm:ss")
```

Differences from QB

- Did not exist in QB. This function appeared in PDS and VBDO:

See also

- **Date Serials**
- **DateSerial**
- **TimeValue**
- **DateValue**

DateValue



Returns a **Date Serial** from a string

Syntax

```
Declare Function DateValue ( ByRef date_string As String ) As Do
```

Usage

```
#include "vbcompat.bi"  
result = DateValue( date_string )
```

Parameters

date_string
the string to convert to a date serial

Return Value

Returns a **Date Serial** from a date string.

Description

The date string must be in the format set in the regional settings of the System.

`DateValue(Date())` will work correctly only if the regional settings specify short date format QB used (mm-dd-yyyy). Consider using the **Now** function expression `Fix(Now())` to obtain the current date as a date serial.

The compiler will not recognize this function unless `vbcompat.bi` or `date.bi` is included.

Example

```
#include "vbcompat.bi"  
  
Dim As Integer v1, v2  
Dim As String s1, s2
```

```

Print "Enter first date: ";
Line Input s1

If IsDate( s1 ) = 0 Then
    Print "not a date"
End
End If

Print "Enter second date: ";
Line Input s2

If IsDate( s2 ) = 0 Then
    Print "not a date"
End
End If

'' convert the strings to date serials
v1 = DateValue( s1 )
v2 = DateValue( s2 )

Print "Number of days between dates is " & Abs( v2

```

Differences from QB

- Did not exist in QB. This function appeared in PDS and VBDO:

See also

- **Date Serials**
- **DateSerial**
- **TimeValue**

Gets the day of the month from a **Date Serial**

Syntax

```
Declare Function Day ( ByVal date_serial As Double ) As Long
```

Usage

```
#include "vbcompat.bi"  
result = Day( date_serial )
```

Parameters

date_serial
the date

Return Value

Returns the day of the month from a variable containing a date in **Date Serial** format.

Description

The compiler will not recognize this function unless `vbcompat.bi` is included.

Example

```
#include "vbcompat.bi"  
  
Dim ds As Double = DateSerial(2005, 11, 28)  
  
Print Format(ds, "yyyy/mm/dd "); Day(ds)
```

Differences from QB

- Did not exist in QB. This function appeared in PDS and VBDO:

See also

- **Date Serials**

Deallocate



Frees previously allocated memory

Syntax

```
Declare Sub Deallocate cdecl ( ByVal pointer As Any Ptr )
```

Usage

```
Deallocate( pointer )
```

Parameters

pointer

the address of the previously allocated buffer.

Description

This procedure frees memory that was previously allocated with `Allocate` returns, *pointer* will be rendered invalid (pointing to an invalid memory, `deallocate` again) will result in undefined behavior.

Calling `Deallocate` on a null pointer induces no action.

`deallocate` is an alias for the C runtime library's `free`, so it's not guaranteed.

Example

The following example shows how to free previously allocated memory:

```
Sub DeallocateExample1()  
    Dim As Integer Ptr integerPtr = Allocate( Len(  
  
    *integerPtr = 420  
    Print *integerPtr
```

```

Deallocate( integerPtr )
integerPtr = 0
End Sub

DeallocateExample1()
End 0

```

Although in this case it is unnecessary since the function immediately gets into. If the function deallocated memory from a pointer that was used in the function call will be rendered invalid, and it is up to the caller to show how to correctly handle this kind of situation, and the next example shows multiple references.

In the following example, a different pointer is used to free previously

```

'' WARNING: "evil" example showing how things should not be done
Sub DeallocateExample2()
  Dim As Integer Ptr integerPtr = Allocate( Len(Integer))
  '' initialize ^^ pointer to new memory

  Dim As Integer Ptr anotherIntegerPtr = integerPtr
  '' initialize ^^ another pointer to the same memory

  *anotherIntegerPtr = 69
  Print *anotherIntegerPtr

  Deallocate( anotherIntegerPtr )
  anotherIntegerPtr = 0

  '' *integerPtr = 420

End Sub

DeallocateExample2()
End 0

```

Note that after the deallocation, *both* pointers are rendered invalid. This is a general rule when working with pointers. As a general rule, only deallocate memory from one (1) pointer currently pointing at it.

```
Function createInteger() As Integer Ptr
    Return Allocate( Len( Integer ) )
End Function

Sub destroyInteger( ByRef someIntegerPtr As Integer Ptr )
    Deallocate( someIntegerPtr )
    someIntegerPtr = 0
End Sub

Sub DeallocateExample3()
    Dim As Integer Ptr integerPtr = createInteger()

    *integerPtr = 420
    Print *integerPtr

    destroyInteger( integerPtr )
    Assert( integerPtr = 0 )
End Sub

DeallocateExample3()
End 0
```

In the program above, a reference pointer in a function is set to null and is used to test if the original pointer is in fact null after the function call. Functions that deallocate the memory they point to do so by reference.

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- [Allocate](#)
- [Reallocate](#)

Declare



Declares a module-level or member procedure

Syntax

```
Declare Sub name [ param_list ]  
Declare Function name [ param_list ] As return_type  
Declare Operator op_symbol param_list [ As return_type ]  
  
Type T  
Declare Constructor [ param_list ]  
Declare Destructor  
Declare Sub name [ param_list ]  
Declare Function name [ param_list ] As return_type  
Declare Operator name [ param_list ] [ As return_type ]  
Declare Property name [ ( [ param_list ] ) ] [ As return_type ]  
End Type
```

Parameters

param_list

Parenthesized comma-separated list of parameters.

return_type

The return type of a **Function**, **Operator**, or **Property** procedure.

name

The name or symbol of the routine.

op_symbol

The name or symbol of an operator.

T

The name of a new user-defined type.

Description

The **declare** statement declares a **Sub**, **Function**, **Operator**, **Constructor**, or **Destructor** routine.

The routine can be referred to in code without seeing its definition, although the **declare** statement introduces a routine, and states that its definition is declared at the top of a source module, called, then defined at the bottom. For example:

A routine's declaration is almost identical to the first line of its definition. The **declare** keyword and has no body. Also, attributes such as **Export** are

FreeBASIC, as QB, does not require the declaration of the functions used in the same file past the point where they are called. This is no longer the case in C++ which must **always** be declared first in the **Type's** body before use. If not, it results in an error.

As every file using a function must have its declaration, declarations are included in the usage of the function by any module that needs it using the `#include` statement.

Example

Module-level Function:

```
' ' declare the function sum which takes two integers
Declare Function sum( As Integer, As Integer ) As Integer

Print "the sum of 420 and 69 is: " & sum( 420, 69 )

' ' define the function sum which takes two integers
Function sum( a As Integer, b As Integer ) As Integer
Return a + b
End Function
```

Type-level Sub:

```
Type my_type
    my_data As Integer
    Declare Sub increment_data( )
End Type

Sub my_type.increment_data( )
    my_data += 1
End Sub

Dim As my_type an_instance

an_instance.my_data = 68
```

```
an_instance.increment_data( )
```

```
Print an_instance.my_data
```

Dialect Differences

- In the *-lang fb* dialect, **ByVal** is the default parameter passing c
- In the *-lang qb* and *-lang deprecated* dialects, **ByRef** is the de
- Type-level **Sub/Function/Operator/Constructor/Destructor**'s are

Differences from QB

- In FreeBASIC, the parameter names are optional.

See also

- **Sub**
- **Function**
- **Operator**
- **Property**
- **Constructor**
- **Destructor**
- **Constructor (Module)**
- **Destructor (Module)**
- **Type**
- **Dim**
- **Alias**

DefByte



Specifies a default data type for a range of variable names

Syntax

```
DefByte start_letter[-end_letter ][, ...]
```

Parameters

start_letter

the first letter in the range

end_letter

the last letter in the range

Description

`DefByte` specifies that variables and arrays which aren't declared with data type - or not declared at all - are implicitly declared of type `Byte` if the first letter of their names matches a certain letter or lies within an inclusive range of letters.

Example

This will make `bNumber` a `Byte` number since its first letter starts with `b`

```
' ' Compile with -lang fblite or qb
#lang "fblite"

DefByte b
Dim bNumber
```

Dialect Differences

- Available in the `-lang fblite` dialect.
- Not available in the `-lang qb` dialect unless referenced with the alias `__Defbyte`.

Differences from QB

- New to FreeBASIC

See also

- [Byte](#)
- [DefInt](#)
- [DefUByte](#)
- [Dim](#)

Specifies a default data type for a range of variable names

Syntax

```
DefDb1 start_letter[-end_letter ][, ...]
```

Parameters

start_letter

the first letter in the range

end_letter

the last letter in the range

Description

`DefDb1` specifies that variables and arrays which aren't declared with a declared at all - are implicitly declared of type **Double** if the first letter matches a certain letter or lies within an inclusive range of letters.

Example

This will make `aNum` a **Double**-precision decimal number since it is in th

```
' ' Compile with -lang fblite or qb
#lang "fblite"
DefDb1 a-d
Dim aNum 'implicit: As Double
Print Len(aNum) ' Prints 8, the number of bytes in
```

Dialect Differences

- Only available in the *-lang qb* and *-lang fblite* dialects.

Differences from QB

- None

See also

- `DefInt`
- `DefSng`
- `Dim`
- `Double`

Preprocessor function to test if a symbol has been defined

Syntax

```
defined (symbol_name)
```

Parameters

symbol_name

Name of the symbol to test

Return Value

Returns non-zero (-1) if the symbol has been defined, otherwise return 0.

Description

Given the symbol name, the `defined()` preprocessor function returns true if the symbol has been defined - or false if the symbol is unknown.

This is used mainly with `#if`.

Similar to `#ifdef` except it allows more than one check to occur because of flexibility.

Example

```
'e.g. - which symbols are defined out of a, b, c,  
  
Const a = 300  
#define b 12  
Dim c As Single  
  
#if defined(a)  
    Print "a is defined"  
#endif  
#if defined(b)
```

```
Print "b is defined"  
#endif  
#if defined(c)  
Print "c is defined"  
#endif  
#if defined(d)  
Print "d is defined"  
#endif
```

Differences from QB

- New to FreeBASIC

See also

- `#define`
- `#macro`
- `#if`
- `#else`
- `#elseif`
- `#endif`
- `#ifdef`
- `#ifndef`
- `#undef`

Specifies a default data type for a range of variable names

Syntax

```
DefInt start_letter[-end_letter ][, ...]
```

Parameters

start_letter
the first letter in the range

end_letter
the last letter in the range

Description

DefInt specifies that variables and arrays which aren't declared with a data type - or not declared at all - are implicitly declared of type **Integer** if the first letter of their names matches a certain letter or lies within an inclusive range of letters.

Example

This will make `iNumber` an **Integer** number since its first letter starts with `i`.

```
' ' Compile with -lang fblite or qb  
  
#lang "fblite"  
  
DefInt i  
Dim iNumber
```

Dialect Differences

- Only available in the **-lang qb** and **-lang fblite** dialects.

Differences from QB

- None

See also

- DefByte
- DefDbl
- DefLng
- Deflongint
- DefShort
- DefSng
- DefStr
- Integer

Specifies a default data type for a range of variable names

Syntax

```
DefLng start_letter[-end_letter ][, ...]
```

Parameters

start_letter

the first letter in the range

end_letter

the last letter in the range

Description

`DefLng` specifies that variables and arrays which aren't declared with a declared at all - are implicitly declared of type **Long** if the first letter of `t` certain letter or lies within an inclusive range of letters.

Example

This will make `lNumber` a **Long** integer number since it starts with `l`.

```
' ' Compile with -lang fblite or qb
#lang "fblite"

DefLng l
Dim lNumber ' implicit: As Long

Print Len(lNumber) ' Displays 4, the number of byt
```

Dialect Differences

- Only available in the **-lang qb** and **-lang fblite** dialects.

Differences from QB

- None

See also

- `DefInt`
- `Defulongint`
- `Dim`
- `LongInt`

Deflongint



Specifies a default data type for a range of variable names

Syntax

```
Deflongint start_letter[-end_letter ][, ...]
```

Parameters

start_letter
the first letter in the range
end_letter
the last letter in the range

Description

Deflongint specifies that variables and arrays which aren't declared with a data type - or not declared at all - are implicitly declared of type **LongInt** if the first letter of their names matches a certain letter or lies within an inclusive range of letters.

Example

This will make `lNumber` a **LongInt** number since its first letter starts with `l`.

```
' ' Compile with -lang fblite  
  
#lang "fblite"  
  
deflongint l  
Dim lNumber
```

Dialect Differences

- Available in the **-lang fblite** dialect.
- Not available in the **-lang qb** dialect unless referenced with the

alias `__Deflongint`.

Differences from QB

- New to FreeBASIC

See also

- `DefInt`
- `Defulongint`
- `Dim`
- `LongInt`

DefShort



Specifies a default data type for a range of variable names

Syntax

```
DefShort start_letter[-end_letter ][, ...]
```

Parameters

start_letter
the first letter in the range
end_letter
the last letter in the range

Description

`DefShort` specifies that variables and arrays which aren't declared with a data type - or not declared at all - are implicitly declared of type `Short` if the first letter of their names matches a certain letter or lies within an inclusive range of letters.

Example

This will make `sNumber` a `Short` number since its first letter starts with `s`

```
' ' Compile with -lang fblite or qb  
  
#lang "fblite"  
  
DefShort s  
Dim sNumber
```

Dialect Differences

- Available in the `-lang fblite` dialect.
- Not available in the `-lang qb` dialect unless referenced with the alias `__Defshort`.

Differences from QB

- New to FreeBASIC
- In QBasic, to make variables default to a 2 byte integer, DEFINT is used.

See also

- [DefInt](#)
- [DefUShort](#)
- [Dim](#)
- [Integer](#)
- [Short](#)

Specifies a default data type for a range of variable names

Syntax

```
DefSng start_letter[-end_letter ][, ...]
```

Parameters

start_letter

the first letter in the range

end_letter

the last letter in the range

Description

`DefSng` specifies that variables and arrays which aren't declared with a data type - or not declared at all - are implicitly declared of type `Single` if the first letter of their names matches a certain letter or lies within an inclusive range of letters.

Example

This will make `sNumber` and `yNumber` a `Single`-precision decimal number since it is in the range of s-z.

```
' ' Compile with -lang fblite or qb  
  
#lang "fblite"  
  
DefSng s-z  
Dim sNumber, yNumber
```

Dialect Differences

- Only available in the `-lang qb` and `-lang fblite` dialects.

Differences from QB

- None

See also

- [DefInt](#)
- [DefDbl](#)
- [Single](#)

Specifies a default data type for a range of variable names

Syntax

```
DefStr start_letter[-end_letter ][, ...]
```

Parameters

start_letter
the first letter in the range

end_letter
the last letter in the range

Description

`DefStr` specifies that variables and arrays which aren't declared with a data type - or not declared at all - are implicitly declared of type `String` if the first letter of their names matches a certain letter or lies within an inclusive range of letters.

Example

This will make `sMessage` a `String` since it starts with `s`.

```
' ' Compile with -lang fblite or qb  
  
#lang "fblite"  
  
DefStr s  
Dim sMessage
```

Dialect Differences

- Only available in the *-lang qb* and *-lang fblite* dialects.

Differences from QB

- None

See also

- DefInt
- DefSng
- DefLng
- DefDbl
- Dim
- String

DefUByte



Specifies a default data type for a range of variable names

Syntax

```
DefUByte start_letter[-end_letter ][, ...]
```

Parameters

start_letter

the first letter in the range

end_letter

the last letter in the range

Description

`DefUByte` specifies that variables and arrays which aren't declared with a data type - or not declared at all - are implicitly declared of type `UByte` if the first letter of their names matches a certain letter or lies within an inclusive range of letters.

Example

This will make `uNumber` a `UByte` number since its first letter starts with

```
' ' Compile with -lang fblite

#lang "fblite"

DefUByte u
Dim uNumber
```

Dialect Differences

- Available in the `-lang fblite` dialect.
- Not available in the `-lang qb` dialect unless referenced with the alias `__Defubyte`.

Differences from QB

- New to FreeBASIC

See also

- [DefByte](#)
- [DefInt](#)
- [Dim](#)
- [UByte](#)

Specifies a default data type for a range of variable names

Syntax

```
DefUInt start_letter[-end_letter ][, ...]
```

Parameters

start_letter
the first letter in the range
end_letter
the last letter in the range

Description

DefUInt specifies that variables and arrays which aren't declared with data type - or not declared at all - are implicitly declared of type **UInteger** if the first letter of their names matches a certain letter or lies within an inclusive range of letters.

Example

This will make `uNumber` a **UInteger** number since its first letter starts with `u`.

```
' ' Compile with -lang fblite  
  
#lang "fblite"  
  
DefInt u  
Dim uNumber
```

Dialect Differences

- Available in the **-lang fblite** dialect.
- Not available in the **-lang qb** dialect unless referenced with the

alias `__Defuint`.

Differences from QB

- New to FreeBASIC

See also

- `DefInt`
- `Dim`
- `UInteger`

Defulongint



Specifies a default data type for a range of variable names

Syntax

```
Defulongint start_letter[-end_letter ][, ...]
```

Parameters

start_letter

the first letter in the range

end_letter

the last letter in the range

Description

`Defulongint` specifies that variables and arrays which aren't declared with a data type - or not declared at all - are implicitly declared of type `ULongInt` if the first letter of their names matches a certain letter or lies within an inclusive range of letters.

Example

This will make `lNumber` a `ULongInt` number since its first letter starts with `l`.

```
' ' Compile with -lang fblite

#lang "fblite"

defulongint l
Dim lNumber
```

Dialect Differences

- Available in the `-lang fblite` dialect.
- Not available in the `-lang qb` dialect unless referenced with the

alias `__Defulongint`.

Differences from QB

- New to FreeBASIC

See also

- `DefInt`
- `Deflongint`
- `Dim`
- `ULongInt`

DefUShort



Specifies a default data type for a range of variable names

Syntax

```
DefUShort start_letter[-end_letter ][, ...]
```

Parameters

start_letter
the first letter in the range
end_letter
the last letter in the range

Description

DefUShort specifies that variables and arrays which aren't declared with a data type - or not declared at all - are implicitly declared of type **ushort** if the first letter of their names matches a certain letter or lies within an inclusive range of letters.

Example

This will make `uNumber` a **UShort** number since its first letter starts with `u`.

```
' ' Compile with -lang fblite  
  
#lang "fblite"  
  
DefUShort u  
Dim uNumber
```

Dialect Differences

- Available in the **-lang fblite** dialect.
- Not available in the **-lang qb** dialect unless referenced with the

alias `__Defushort`.

Differences from QB

- New to FreeBASIC

See also

- [DefInt](#)
- [DefShort](#)
- [Dim](#)
- [UShort](#)

Operator Delete



Operator to delete data allocated with the `New` operator

Syntax

```
Declare Operator Delete ( buf As Any Ptr )  
Declare Operator delete[] ( buf As Any Ptr )
```

Usage

```
Delete buf  
or  
Delete[] buf
```

Parameters

buf

A pointer to memory that has been allocated by `New` or `New[]` (a typed accordance to the data type to delete).

Description

`Delete` is used to destroy and free the memory of an object created with `New`. A destructor will be called. `Delete` should only be used with addresses returned from `New`.

The array version of `Delete`, `Delete[]`, is used to destroy an array of objects. Destructors will be called here as well.

`Delete` must be used with addresses returned from `New`, and `Delete[]` must match the different versions of the operators.

After the memory is deleted, the *buf* pointer will be pointing at invalid memory. The same pointer value leads to undefined behaviour. It may be a good idea to set the pointer to null in order to guard against later code using it accidentally, since null pointers are easier to debug.

Calling `Delete` on a null pointer induces no action.

Example

```

Type Rational
  As Integer numerator, denominator
End Type

' Create and initialize a Rational, and store its
Dim p As Rational Ptr = New Rational(3, 4)

Print p->numerator & "/" & p->denominator

' Destroy the rational and give its memory back to
Delete p

' Set the pointer to null to guard against future
p = 0

```

```

' Allocate memory for 100 integers, store the addr
Dim p As Integer Ptr = New Integer[100]

' Assign some values to the integers in the array.
For i As Integer = 0 To 99
  p[i] = i
Next

' Free the entire integer array.
Delete[] p

' Set the pointer to null to guard against future
p = 0

```

Dialect Differences

- Only available in the *-lang fb* dialect.

Differences from QB

- New to FreeBASIC

See also

- [New](#)
- [Deallocate](#)

Destructor



Called automatically when a class or user defined type goes out of scope or is destroyed

Syntax

```
Type typename  
field declarations  
Declare Destructor ( )  
End Type  
  
Destructor typename ( ) [ Export ]  
statements  
End Destructor
```

Parameters

typename
name of the **Type** of **Class**

Description

The destructor method is called when a user defined **Type** or **Class** variable goes out of scope or is destroyed explicitly with the **Delete** operator.

typename is the name of the type for which the **Destructor** method is declared and defined. Name resolution for *typename* follows the same rules as procedures when used in a **Namespace**.

The **Destructor** method is passed a hidden **This** parameter having the same type as *typename*.

The destructor in a type is called before the destructors on any of its fields. Therefore, all fields are accessible with the hidden **This** parameter in the destructor body.

Only one destructor may be declared and defined per type.

Since the **End** statement does not close any scope, object destructors

will not automatically be called if the **End** statement is used to terminate the program.

Destructor can be also called directly from the *typename* instance like the other member methods (**Sub**) and with the same syntax, i.e. using member access operator, e.g. *obj.Destructor()*. The object, and all its members, are assumed to be constructed and in a valid state, otherwise its effects are undefined and may cause crashes. This syntax is useful in cases where *obj* has been constructed manually, e.g. with *obj.Constructor()* or **Placement New**.

Example

```
Type T
  value As ZString * 32
  Declare Constructor ( init_value As String )
  Declare Destructor ( )
End Type

Constructor T ( init_value As String )
  value = init_value
  Print "Creating: "; value
End Constructor

Destructor T ( )
  Print "Destroying: "; value
End Destructor

Sub MySub
  Dim x As T = ("A.x")
End Sub

Dim x As T = ("main.x")

Scope
  Dim x As T = ("main.scope.x")
End Scope
```

MySub

Output:

```
Creating: main.x  
Creating: main.scope.x  
Destroying: main.scope.x  
Creating: A.x  
Destroying: A.x  
Destroying: main.x
```

Dialect Differences

- Object-related features are supported only in the *-lang fb* dialect

Differences from QB

- New to FreeBASIC

See also

- **Class**
- **Constructor**
- **Delete**
- **Destructor (Module)**
- **Type**

Destructor (Module)



Specifies execution of a procedure at program termination

Syntax

```
[Public | Private] Sub identifier [Alias "external_identifier"]  
[Static]  
{ procedure body }  
End Sub
```

Description

Defines a procedure to be automatically called from a compiled program generated by the compiler and is executed when the program terminates. Destructors defined as destructors may be used the same way as ordinary procedures called from within module-level code, as well as other procedures.

The procedure must have an empty parameter list. A compile-time error **DESTROY** keyword is used in a Sub definition having one or more parameters. In overloaded procedures, only one (1) destructor may be defined because having multiple Subs which take no arguments.

In a single module, destructors normally execute in the order in which

The *priority* attribute, an integer between 101 and 65535, can be used to control the order in which destructors are executed. The value of *priority* has no specific meaning other than the number with other destructor priorities. 101 is the lowest priority and 65535 is the highest. Destructors having a *priority* attribute are executed after destructors with a *priority* value of 65535 is the same as not assigning a priority value.

A module may define multiple destructor procedures. Destructors may be defined in more than one module. All procedures defined with the syntax shown above are a list of procedures to be called during the program's termination.

The order in which destructors defined in multiple modules are executed is not guaranteed. Therefore, special care should be taken when using destructors in a secondary module also defining a destructor. In such a case it is advised to use a destructor that explicitly calls termination procedures in multiple module termination of the application.

Destructors will be called if the program terminates normally or if error the program terminates abnormally.

Example

```
Sub pauseonexit Destructor

    '' If the program reaches the end, or aborts w
    '' it will run this destructor before closing

    Print "Press any key to end the program..."
    Sleep

End Sub

Dim array(0 To 10, 0 To 10) As Integer
Dim As Integer i = 0, j = 11

'' this next line will cause the program to abort
'' error if you compile with array bounds checking
exx ...)
Print array(i, j)
```

Differences from QB

- New to FreeBASIC

See also

- **Destructor (Class)**
- **Constructor (Module)**
- **Sub**

Declares a variable

Syntax

```
Dim [Shared] name1 As DataType [, name2 As DataType, ...]
```

Or

```
Dim [Shared] As DataType name1 [, name2, ...]
```

Arrays:

```
Dim name ( [lbound To] ubound [, ...] ) As DataType
```

```
Dim name ( Any [, Any...] ) As DataType
```

```
Dim name ( ) As DataType
```

Initializers:

```
Dim scalar_symbol As DataType = expression | Any
```

```
Dim array_symbol (arraybounds) As DataType = { expression [, ...
```

```
Dim udt_symbol As DataType = ( expression [, ...] ) | Any
```

Description

Declares a variable by name and reserves memory to accommodate i

Variables must be declared before they can be used in the *-lang fb* di other dialects. Only in the *-lang qb* and *-lang fbLite* dialects variables in such a case they are called implicit variables.

`Dim` can be used to declare and assign variables of any of the support enumerations.

Depending on where and how a variable or array is declared can char *Storage Classes*.

More than one variable may be declared in a single `Dim` statement by comma.

```
' ' Variable declaration examples  
  
' ' One variable per DIM statement  
Dim text As String
```

```
Dim x As Double
```

```
' More than one variable declared, different data  
Dim k As Single, factor As Double, s As String
```

```
' More than one variable declared, all same data  
Dim As Integer mx, my, mz ,mb
```

```
' Variable having an initializer  
Dim px As Double Ptr = @x
```

Explicit Variables with Implicit Data Types

In the *-lang qb* and *-lang fblite* dialects, even if the variable is declared without an explicit data type, the compiler will infer the data type based on the variable name. In the *-lang qb* dialect, the default data type is `Integer`. In the *-lang fblite* dialect, the default data type is `Double`. The default data type can be overridden by use of the `Def###` statements. (for example, `DefInt, D`

```
' Compile with -lang qb  
'$lang: "qb"  
  
' All variables beginning with A through N default to Integer  
' All other variables will default to the SINGLE data type  
DefInt I-N  
  
' I and J are INTEGERS  
' X and Y are SINGLES  
' T$ is STRING  
' D is DOUBLE  
  
Dim I, J, X, Y, T$, D As Double
```

Arrays

As with most BASIC dialects, FreeBASIC supports arrays with indexes bound. In the syntaxes shown, *lbound* refers to the lower bound, or the bound, or the largest index. If a lower bound is not specified, it is assumed **Base** is used.

```
Const upperbound = 10

' Declare an array with indexes ranging from 0 to
' for a total of (upperbound + 1) indexes.
Dim array(upperbound) As Single
```

Multidimensional arrays can be declared as well, and are stored in this last index are contiguous (row-major order).
The maximum number of dimensions of a multidimensional array is 8.

```
' declare a three-dimensional array of single
' precision floating-point numbers.
Dim array(1 To 2, 6, 3 To 5) As Single

' The first dimension of the declared array
' has indices from 1 to 2, the second, 0 to 6,
' and the third, 3 to 5.
```

For more information on arrays see [Arrays Overview](#).

If the values used with **Dim** to declare the dimensions of an array are **Static** (unless **Option Dynamic** is specified), while using one or more **Var** array makes it variable length, even if **Option Static** is in effect.

Arrays can be declared as variable length in several ways: Using **Dim** with **Var**, using **Dim** with indexes that are variables or using the keyword **ReDim**, or declaring it past the metacommand **\$Dynamic**. Variable length arrays:

Arrays declared with `Dim` having constant indexes and not preceded (resizable at runtime) and can use initializers.

The upper bound can be an ellipsis (`...`, 3 dots). This will cause to up on the number of elements found in the initializer. When ellipsis is used, and it may not be `Any`. See the [Ellipsis](#) page for a short example.

See also [Fixed-Length Arrays](#) and [Variable-Length Arrays](#).

Initializers

Arrays, variables, strings, and user defined types (UDTs) are initialized by default when they are created.

To avoid the overhead of default variable initialization, the `Any` initializer to only reserve the place for the variable in memory but not initialize it. In this case the programmer should not make assumptions about the initial value.

Fixed-length arrays, variables, zstrings and UDTs may be given a value following the variable declaration with an initializer. Note the difference: Arrays, variables and UDTs are initialized as they would in a normal assignment. The `=>` sign can be used, allowing to avoid the declaration resembling an array of strings.

Array values are given in comma-delimited values enclosed by curly brackets. For variables, comma delimited values enclosed by parenthesis. These methods of initialization can be used one another for complex assignments. Nesting allows for arrays of arrays.

```
' ' Declare an array of 2 by 5 elements
' ' and initialize
Dim array(1 To 2, 1 To 5) As Integer => {{1, 2, 3,
```

```
' ' declare a simple UDT
Type mytype
    var1 As Double
```

```

    var2 As Integer
End Type

' declare a 3 element array and initialize the fi
' 2 mytype elements
Dim myvar(0 To 2) As mytype => {(1.0, 1), (2.0, 2)}

```

For module-level, fixed-length, or global variables, initialized values m will report a compile-time error if otherwise.

Note: Initializing UDT's with strings is not supported at this time. Initial string is not valid.

Explicit Variables with Type Suffixes

In the *-lang qb* and *-lang fblite* dialects, the data type of a variable m).

```

' Compile with -lang qb or fblite

'$lang: "qb"

' A string variable using the $ type suffix
Dim strVariable$

' An integer variable using the % type suffix
Dim intVariable%

' A long variable using the & type suffix
Dim lngVariable&

' A single precision floating point variable usir
Dim sngVariable!

' A double precision floating point variable usir
Dim dblVariable#

```

Example

```
Dim a As Byte
Dim b As Short
Dim c As Integer
Dim d As LongInt
Dim au As UByte
Dim bu As UShort
Dim cu As UInteger
Dim du As ULongInt
Dim e As Single
Dim f As Double
Dim g As Integer Ptr
Dim h As Byte Ptr
Dim s1 As String * 10    '' fixed length string
Dim s2 As String        '' variable length string
Dim s3 As ZString Ptr   '' zstring

s1 = "Hello World!"
s2 = "Hello World from FreeBASIC!"
s3 = Allocate( Len( s2 ) + 1 )
*s3 = s2

Print "Byte: "; Len(a)
Print "Short: "; Len(b)
Print "Integer: "; Len(c)
Print "Longint: "; Len(d)
Print "UByte: "; Len(au)
Print "UShort: "; Len(bu)
Print "UInteger: "; Len(cu)
Print "ULongint: "; Len(du)
Print "Single: "; Len(e)
Print "Double: "; Len(f)
Print "Integer Pointer: "; Len(g)
Print "Byte Pointer: "; Len(h)
Print "Fixed String: "; Len(s1)
```


- **ReDim**
- **Preserve**
- **Shared**
- **Static**
- **Erase**
- **LBound**
- **UBound**
- **... (Ellipsis)**
- **Any**

Searches for and returns information about an item in the filesystem; pe

Syntax

```
# Include "dir.bi"
```

```
Declare Function Dir ( ByRef item_spec As Const String, ByVal at  
Declare Function Dir ( ByRef item_spec As Const String, ByVal at  
Declare Function Dir ( ByVal attrib_mask As Integer = fbNormal,  
Declare Function Dir ( ByVal attrib_mask As Integer = fbNormal,
```

Usage

```
result = Dir( item_spec, [ attrib_mask ], out_attrib )  
result = Dir( item_spec [, [ attrib_mask ] [, p_out_attrib ] ] )  
result = Dir( out_attrib )  
result = Dir( [ p_out_attrib ] )
```

Parameters

item_spec

The pattern to match an item's name against.

attrib_mask

The bit mask to match an item's attributes against.

out_attrib

Reference to a bit mask that's assigned each of the found item's attrib

p_out_attrib

Pointer to a bit mask that's assigned each of the found item's attribute

Return Value

If no item matching the name *item_spec* or the attribute mask *attrib_1* (or **p_out_attrib*) is assigned the attribute mask of the item, and the

Description

If *item_spec* contains an absolute path, then the first procedure search
Otherwise, it searches relative to the current directory (see [CurDir](#)). In
out_attrib is assigned with the attribute flags of the item, and the nar

item_spec may include an asterisk (*, for matching any adjacent characters such item. If found, subsequent calls with *item_spec* omitted, or set to omitted from these subsequent calls, the procedure searches for item

The second syntax behaves the same as `Dir(item_spec, attrib_mask`

The third syntax behaves the same as `Dir("", , out_attrib)`.

The fourth syntax behaves the same as `Dir("", , *p_out_attrib)`.

File Attributes:

Files and directories and other items can be said to possess so-called attributes and the file system it uses.

The following defined constants are used as bit-flags in *attrib_mask* a metadata that the returned files are **allowed** to have. For example, `fbReadOnly` will be matched. (`fbReadOnly` or `fbDirectory`) will allow read-only directories. More powerful filtering can be done by checking the returned *out_attrib*

```
# define fbReadOnly &h01;
```

The item cannot be written to or deleted.

DOS & Windows: The item has the "read-only" attribute set.

Linux: The item has no write permissions associated with the current item.

```
# define fbHidden &h02;
```

The item is hidden in ordinary directory listings.

DOS & Windows: The item has the "hidden" attribute set.

Linux: The item's name has a period (.) as the first character.

```
# define fbSystem &h04;
```

The item is used almost exclusively by the system.

DOS & Windows: The item has the "system" attribute set.

Linux: The item is either a character device, block device, named pipe,

```
# define fbDirectory &h10;
```

The item is a directory. Includes the current (.) and parent (..) directories.

DOS & Windows & Linux: The item is a directory.

```
# define fbArchive &h20;
```

The item may be backed up after some automated operations.

DOS & Windows: The item has the "archive" attribute set (automatic backup).

Linux: The item is not a directory; typical filesystems do not support this attribute.

```
# define fbNormal (fbReadOnly or fbArchive)
The item is read-only or "archived".
```

(If `attrib_mask` does not include `fbArchive`, then `Dir` may widen the ch

Items found having no attributes are **always** matched, regardless of th
example, `fbArchive` or `fbDirectory` will match against archived files,

In general it is not possible to use `attrib_mask` to include a file/folder v
directories while excluding read-only files (unless the files also have o

Example

```
#include "dir.bi" 'provides constants to use for t

Sub list_files (ByRef filespec As String, ByVal attrib as Integer)
    Dim As String filename = Dir(filespec, attrib)
    Do While Len(filename) > 0 ' If len(filename)
        Print filename
        filename = Dir()
    Loop
End Sub

Print "directories:"
list_files "*", fbDirectory

Print
Print "archive files:"
list_files "*", fbArchive
```

Example

```
' Example of using DIR function and retrieving at
#include "dir.bi" '' provides constants to match t
```

```

'' set input attribute mask to allow files that are
Const attrib_mask = fbNormal Or fbHidden Or fbSystem

Dim As UInteger out_attr '' unsigned integer to hold attributes

Dim As String fname '' file/directory name returned by Dir
Dim As Integer filecount, dircount

fname = Dir("*.**", attrib_mask, out_attr) '' Get first file

Print "File listing in " & CurDir & ":"

Do Until Len(fname) = 0 '' loop until Dir returns empty string
    If (fname <> ".") And (fname <> "..") Then '' skip "." and ".."
        Print fname,

        If (out_attr And fbDirectory) <> 0 Then
            Print "- directory";
            dircount += 1
        Else
            Print "- file";
            filecount += 1
        End If

        If (out_attr And fbReadOnly) <> 0 Then Print "  read-only";
        If (out_attr And fbHidden) <> 0 Then Print "  hidden";
        If (out_attr And fbSystem) <> 0 Then Print "  system";
        If (out_attr And fbArchive) <> 0 Then Print "  archive";
        Print

    End If

    fname = Dir(out_attr) '' find next name/attribute
Loop

Print

```

```
Print "Found " & filecount & " files and " & dircc
```

Platform Differences

- Linux requires the *filename* case to match the real name of the
- Path separators in Linux are forward slashes /. Windows uses
- In DOS, the attrib mask value of &h37; (&h3F; usually works also if current directory is not the main directory.

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- Not found in QBasic but present in Visual Basic. The *out_attr*

See also

- `Open`
- `CurDir`
- `MkDir`
- `Rmdir`

Control flow statement for looping.

Syntax

```
Do [ { Until | While } condition ]  
  [ statement block ]  
Loop
```

```
Do  
  [ statement block ]  
Loop [ { Until | While } condition ]
```

Differences from QB

- None

See also

- [Do...Loop](#)

Do...Loop



Control flow statement for looping

Syntax

```
Do [ { Until | While } condition ]  
  [ statement block ]  
Loop  
or  
Do  
  [ statement block ]  
Loop [ { Until | While } condition ]
```

Description

The **Do** statement executes the statements in the following *statement block*

If **Until** is used, the **Do** statement stops repetition of the *statement block* loop if *condition* evaluates to false. If both *condition* and either **Until**

If an **Exit Do** statement is encountered inside the *statement block*, the statement. If a **Continue Do** statement is encountered, the rest of the *s*

In the first syntax, the *condition* is checked when the **Do** statement is second syntax, *condition* is initially checked *after* the *statement block* once.

condition may be any valid expression that evaluates to False (zero)

Example

In this example, a **Do** loop is used to count the total number of odd num

```
Dim As Integer n = 1  
Dim As Integer total_odd = 0  
Do Until( n > 10 )  
  If( ( n Mod 2 ) > 0 ) Then total_odd += 1  
  n += 1  
Loop  
Print "total odd numbers: " ; total_odd
```

```
End 0
```

Here, an infinite DO loop is used to count the total number of evens. The loop if and when $n > 10$ becomes true:

```
Dim As Integer n = 1
Dim As Integer total_even = 0
Do
    If( n > 10 ) Then Exit Do

    If( ( n Mod 2 ) = 0 ) Then total_even += 1
    n += 1
Loop
Print "total even numbers: " ; total_even

End 0
```

Dialect Differences

- In the *-lang qb* and *-lang fblite* dialects, variables declared inside loops are not automatically destroyed.
- In the *-lang fb* and *-lang deprecated* dialects, variables declared inside loops are not automatically destroyed.

Differences from QB

- None

See also

- `Continue`
- `Exit`
- `For...Next`
- `While...Wend`

Double



Standard data type: 64 bit floating point

Syntax

```
Dim variable As Double
```

Description

Double is a 64-bit, floating-point data type used to store more precise decimal numbers. They can hold positive values in the range $4.940656458412465e-324$ to $1.797693134862316e+308$, or negative value in the range $-4.940656458412465e-324$ to $-1.797693134862316e+308$, or zero (0). They contain at most 53 bits of precision, or about 15 decimal digits.

Doubles have a greater range and precision than **singles**, they still have limited accuracy which can lead to significant inaccuracies if not used properly. They are dyadic numbers - i.e. they can only accurately hold multiples of powers of two, which will lead to inaccuracies in most base-10 fractions.

Example

```
'Example of using a double variable.  
  
Dim a As Double  
a = 1.985766472453666  
Print a  
  
Sleep
```

Differences from QB

- None

See also

- `single` Less precise float type
- `Cdbl`
- Table with variable types overview, limits and suffixes

Statement for sequenced pixel plotting

Syntax

```
Draw [target,] cmd
```

Parameters

target

the buffer to draw on

cmd

a string containing the sequence of commands

Description

Drawing will take place onto the current work page set via [ScreenSet](#) *target* [Get/Put](#) buffer if specified.

The **Draw** statement can be used to issue several drawing commands it is useful to quickly draw figures. The command string accepts the following commands:

Commands to plot pixels:

Command	Description
	Commands to plot pixels:
B	Optional prefix: move but do not draw.
N	Optional prefix: draw but do not move.
M <i>x,y</i>	Move to specified screen location. if a '+' or '-' sign precedes <i>x</i> , movement is relative to current cursor position. <i>x</i> 's sign has no effect on the sign of <i>y</i> .
U [<i>n</i>]	Move <i>n</i> units up. If <i>n</i> is omitted, 1 is assumed.
D [<i>n</i>]	Move <i>n</i> units down. If <i>n</i> is omitted, 1 is assumed.
L [<i>n</i>]	Move <i>n</i> units left. If <i>n</i> is omitted, 1 is assumed.
R [<i>n</i>]	Move <i>n</i> units right. If <i>n</i> is omitted, 1 is assumed.
E [<i>n</i>]	Move <i>n</i> units up and right. If <i>n</i> is omitted, 1 is assumed.
F [<i>n</i>]	Move <i>n</i> units down and right. If <i>n</i> is omitted, 1 is assumed.
G [<i>n</i>]	Move <i>n</i> units down and left. If <i>n</i> is omitted, 1 is assumed.
H [<i>n</i>]	Move <i>n</i> units up and left. If <i>n</i> is omitted, 1 is assumed.

	Commands to color:
C n	Changes current foreground color to n.
P p,b	PAINTs (flood fills) region of border color b with color p.
	Commands to scale and rotate:
S n	Sets the current unit length, default is 4. A unit length of 4 is equal to 1 pixel.
A n	Rotate n*90 degrees (n ranges 0-3).
TA n	Rotate n degrees (n ranges 0-359).
	Extra commands:
X p	Executes commands at p, where p is a STRING PTR.

Commands to set the color, size and angle will take affect all subsequent operations.

Example

Screen 13

```
'Move to (50,50) without drawing
Draw "BM 50,50"
```

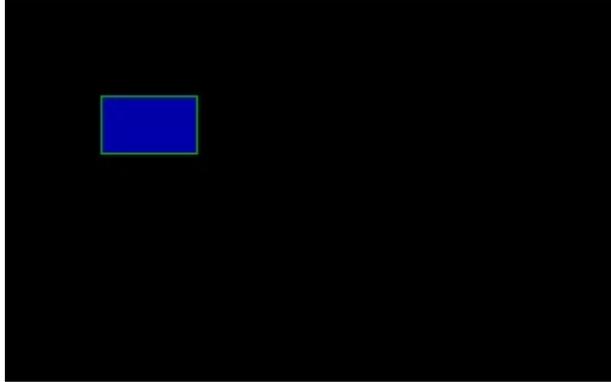
```
'Set drawing color to 2 (green)
Draw "C2"
```

```
'Draw a box
Draw "R50 D30 L50 U30"
```

```
'Move inside the box
Draw "BM +1,1"
```

```
'Flood fill with color 1 (blue) up to border color
Draw "P 1,2"
```

```
Sleep
```



```
' ' Draws a flower on-screen

Dim As Integer i, a, c
Dim As String fill, setangle

' ' pattern for each petal
Dim As Const String petal = _
    _ ("X" & VarPtr(setangle)) _ ' link to angle-
setting string
    _ & "C15" _ ' set outline color (white)
    _ & "M+100,+10" _ ' draw outline
    _ "M +15,-10" _
    _ "M -15,-10" _
    _ "M-100,+10" _
    _ & "BM+100,0" _ ' move inside pet
    _ & ("X" & VarPtr(fill)) _ ' flood-
fill petal (by linking to fill string)
    _ & "BM-100,0" _ ' move back out

' ' set screen
ScreenRes 320, 240, 8
```

```

'' move to center
Draw "BM 160, 120"

'' set initial angle and color number
a = 0: c = 32

For i = 1 To 24

    '' make angle-setting and filling command string
    setangle = "TA" & a
    fill = "P" & c & ",15"

    '' draw the petal pattern, which links to angle
    setting and filling strings
    Draw petal

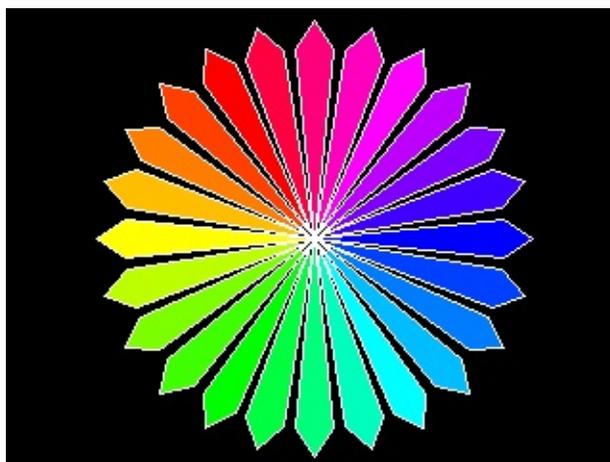
    '' short delay
    Sleep 100

    '' increment angle and color number
    a += 15: c += 1

Next i

Sleep

```



Differences from QB

- *target* is new to FreeBASIC
- QB used the special pointer keyword `VARPTR$` with the `x` `y` `cc`
- FB does not currently allow sub-pixel movements: all movements rounded to the nearest integer coordinate.

See also

- [Draw String](#)
- [Screen \(Graphics\)](#)
- [VarPtr](#)
- [Paint](#)

Draw String



Graphics statement to render text to an image or screen.

Syntax

```
Draw String [buffer,] [STEP] (x, y), text [,color [, font [, met
```

Usage

```
Draw String [buffer,] [STEP] (x, y), text [, color]
```

```
Draw String [buffer,] [STEP] (x, y), text , , font [, method [,
```

```
Draw String [buffer,] [STEP] (x, y), text , , font, Custom, blen
```

Parameters

buffer

the sprite to draw the string on. If this is not supplied, it will be drawn to the target surface.
STEP

use relative coordinates. If STEP is added, the x and y coordinates are

x, *y*

the horizontal / vertical position to draw to, relative to the top left hand corner of the target surface.
The top left corner of the text will be drawn at this position.

text

the string containing the text to draw

color

if no font is supplied, this allows you to choose the color of the text. If

if a font is supplied, *color* is ignored, and the font itself specifies the color.

font

an image buffer containing a custom font. If no font is supplied, the standard font is used.
the following parameters are ignored.

method | **Custom**

specifies how the font characters are drawn on top of the target surface. The following methods are allowed, with the only difference that the default method is **Transparent** for standard fonts.

alpha

alpha value, ranging 0-255. This parameter only applies to the **Add** or **Blend** methods.

blender

custom blender function for the **Custom** drawing method; see **Put (Graphics)**. This parameter only applies to the **Custom** method.

parameter

optional **Pointer** to be passed to the custom blender function; if omitted, the target surface is used.

Description

This graphics keyword prints a string to the screen with pixel position and supplied font. **Draw String** does not update any text or graphics cursor and returns and other special characters have no special behavior in **Draw**

In graphics mode, this function provides a flexible alternative to **Print**.

- **Draw String** can print text to any coordinate on the screen, while **Print Locate**.

- **Print** will override the background behind the text with the current background; the pixels in the background untouched.

- Like **Put**, **Draw String** has several different methods for printing text,

- **Draw String** isn't limited to a single character set: it is possible to sup

Note: If a custom font isn't supplied, **Draw String** will default to the standard font dictated by **width.method** - if passed - will be ignored, and the text will be printed on the background.

The custom font format:

The font is stored in a standard **Get/Put** buffer; the font has to be stored in a buffer of a certain depth, otherwise **Draw String** will bump out with an illegal function call.

The first line of pixels in the font buffer holds the header of the font, or the font header version; currently this must be 0. The second byte gives the font height; the third byte gives the ascii code of the last supported character. The contents of these two bytes will be the contents of these two bytes.

Next comes the width of each of the supported characters, each in a byte from 32 to 127 (inclusive), the header would have the first three bytes as the widths of the corresponding chars.

The font height is obtained by subtracting 1 from the buffer height, the header, the remaining lines define the glyphs' layout. The buffer must contain character sprites in the same row, one after another.

Example

This gives an example of basic **Draw String** usage: it uses it to print "I

```

Const w = 320, h = 200 '' screen dimensions

Dim x As Integer, y As Integer, s As String

'' Open a graphics window
ScreenRes w, h

'' Draw a string in the centre of the screen:

s = "Hello world"
x = (w - Len(s) * 8) \ 2
y = (h - 1 * 8) \ 2

Draw String (x, y), s

'' Wait for a keypress before ending the program
Sleep

```



This example shows you how to create and use your own custom font to create the glyphs.

```

'' Define character range
Const FIRSTCHAR = 32, LASTCHAR = 127

Const NUMCHARS = (LASTCHAR - FIRSTCHAR) + 1
Dim As UByte Ptr p, myFont
Dim As Integer i

'' Open a 256 color graphics screen (320*200)

```

```

ScreenRes 320, 200, 8

'' Create custom font into PUT buffer
myFont = ImageCreate(NUMCHARS * 8, 9)

'' Put font header at start of pixel data

#ifdef ImageInfo '' older versions of FB don't ha
p = myFont + IIf(myFont[0] = 7, 32, 4)
#else
ImageInfo( myFont, , , , , p )
#endif

p[0] = 0
p[1] = FIRSTCHAR
p[2] = LASTCHAR

'' PUT each character into the font and update wi
For i = FIRSTCHAR To LASTCHAR

'' Here we could define a custom width for eac
'' a fixed width of 8 since we are reusing the
p[3 + i - FIRSTCHAR] = 8

'' Create character onto custom font buffer by
Draw String myFont, ((i - FIRSTCHAR) * 8, 1),

Next i

'' Now the font buffer is ready; we could save it
Rem BSave "myfont.bmp", myFont

'' Here we draw a string using the custom font
Draw String (10, 10), "ABCDEFGHJKLMNOPQRSTUVWXYZ"
Draw String (10, 26), "abcdefghijklmnopqrstuvwxy"
Draw String (66, 58), "Hello world!", , myFont

'' Free the font from memory, now we are done with

```

ImageDestroy myFont

Sleep



Differences from QB

- New to FreeBASIC

See also

- (Print | ?)
- Draw
- ImageCreate
- ImageDestroy
- ImageInfo
- Put (Graphics)
- Width

Unloads a dynamic link library from memory

Syntax

```
Declare Sub DyLibFree ( ByVal library As Any Pointer )
```

Usage

```
DyLibFree( library )
```

Parameters

library

The handle of a library to unload.

Description

`DyLibFree` is used to release at runtime libraries previously linked to your program with `DyLibLoad`. The argument is the handle to the library returned by `DyLibLoad`.

Example

See the dynamic loading example on the [Shared Libraries](#) page.

Platform Differences

- Dynamic link libraries are not available in DOS, as the OS doesn't support them.

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__DyLibFree`.

Differences from QB

- New to FreeBASIC

See also

- **DyLibSymbol**
- **DyLibLoad**
- **Export**

Loads to a Dynamic Link Library (DLL) into memory at runtime

Syntax

```
Declare Function DyLibLoad ( ByRef filename As String ) As Any  
Pointer
```

Usage

```
result = DyLibLoad ( filename )
```

Parameters

filename

A **String** containing the filename of the library to load.

Return Value

The **Pointer** handle of the library loaded. Zero on error

Description

DyLibLoad is used to link at runtime libraries to your program. This function does the link and returns a handle that must be used with **DyLibSymbol** when calling a function in the library and with **DyLibFree** when releasing the library.

Example

See the dynamic loading example on the **Shared Libraries** page.

Platform Differences

- Dynamic link libraries are not available in DOS, as the OS doesn't support them.

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__DyLibload`.

Differences from QB

- New to FreeBASIC

See also

- [DyLibSymbol](#)
- [DyLibFree](#)
- [Export](#)

DyLibSymbol



Returns the address of a function or variable in a dll

Syntax

```
Declare Function DyLibSymbol ( ByVal library As Any Ptr, ByRef  
symbol As String ) As Any Ptr  
Declare Function DyLibSymbol ( ByVal library As Any Ptr, ByVal  
symbol As Short ) As Any Ptr
```

Usage

```
result = DyLibSymbol ( library, symbol )
```

Parameters

library

The **Any Ptr** handle of a DLL returned by **DyLibLoad**

symbol

A **String** containing name of the function, or variable in the library to return the address of. In Windows only, can also be a **Short** containing the ordinal of the function/variable.

Return Value

A **Pointer** to the function or variable in the library.

If the function fails, the return value is 0.

Description

DyLibSymbol returns a pointer to the variable or function named *symbol* in the dll pointed by *libhandle*. *libhandle* is obtained by loading the dll with **DyLibLoad**. The symbol must have been **Exported** in the dll.

If *libhandle* is 0, the symbol is searched in the current executable or dll.

If using **cdecl** functions, only the name of the procedure needs to be specified. If dynamically linking to a function created using STDCALL (default in windows), then the function must be decorated. To decorate

a function, use its name, '@', then the number of bytes passed as arguments. For instance if the function `foo` takes 3 integer arguments, the decorated function would be `'FOO@12'`. Remember, without an explicit **Alias**, the procedure name will be uppercase.

If linking to a dll created in Visual C++(tm), decoration need not be used. For GCC, decoration is needed.

Note: The `dylibsymbol`, if failing, will attempt to automatically decorate the procedure, from `@0` to `@256`, in 4 byte increments.

Example

See the dynamic loading example on the **Shared Libraries** page.

Platform Differences

- Dynamic link libraries are not available in DOS ,as the OS doesn't support them.
- Ordinals are not supported on Linux, 0 is always returned.

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__DyLibsymbol`.

Differences from QB

- New to FreeBASIC

See also

- **DyLibLoad**
- **Export**

Control flow statement for conditional branching

Syntax

```
If expression Then statement(s) [Else statement(s)]  
or  
If expression Then : statement(s) [Else statement(s)] : End If  
or  
If expression Then  
statement(s)  
[ ElseIf expression Then ]  
statement(s)  
[ Else ]  
statement(s)  
End If
```

Differences from QB

- None

See also

- [If...Then](#)

Control flow statement for conditional branching

Syntax

```
If expression Then  
statement(s)  
[ ElseIf expression Then ]  
statement(s)  
[ Else ]  
statement(s)  
End If
```

Differences from QB

- None

See also

- [If...Then](#)

Specifies character format of a text file

Syntax

Open *filename* for {Input|Output|Append} **Encoding** "utf-8"|"utf-16"|"utf-32"|"ascii" as [#]*filenum*

Parameters

filename for {Input|Output|Append}

file name to open for **Input**, **Output**, or **Append**

Encoding "utf-8"|"utf-16"|"utf-32"|"ascii"

indicates encoding type for the file

filenum

unused file number to associate with the open file

Description

Encoding specifies the format for an Unicode text file, so **winput #** and correct encoding. If omitted from an **open** statement, "ascii" encoding is used.

Only little endian character encodings are supported at the moment.

- "utf8",
- "utf16"
- "utf32"
- "ascii" (the default)

Example

```
' ' This example will:
' ' 1) Write a string to a text file with utf-16 encoding
' ' 2) Display the byte contents of the file
' ' 3) Read the text back from the file
' '
' ' WSTRING's will work as well but STRING has been
' ' used in this example since not all consoles support
```

```

'' printing WSTRING's.

'' The name of the file to use in this example
Dim f As String
f = "sample.txt"

''
Scope
  Dim s As String
  s = "FreeBASIC"

  Print "Text to write to " + f + ":"
  Print s
  Print

  '' open a file for output using utf-16 encoding
  '' and print a short message
  Open f For Output Encoding "utf-16" As #1

  '' The ascii string is converted to utf-16
  Print #1, s
  Close #1
End Scope

''
Scope
  Dim s As String, n As Integer

  '' open the same file for binary and read all th
  Open f For Binary As #1
  n = LOF(1)
  s = Space( n )
  Get #1,,s
  Close #1

  Print "Binary contents of " + f + ":"
  For i As Integer = 1 To n
    Print Hex( Asc( Mid( s, i, 1 ) ), 2); " ";
  Next

```

```

Print
Print

End Scope

''
Scope
  Dim s As String

  '' open a file for input using utf-16 encoding
  '' and read back the message
  Open f For Input Encoding "utf-16" As #1

  '' The ascii string is converted from utf-16
  Line Input #1, s
  Close #1

  '' Display the text
  Print "Text read from " + f + ":"
  Print s
  Print
End Scope

```

Output:

```

Text to write to sample.txt:
FreeBASIC

Binary contents of sample.txt:
FF FE 46 00 72 00 65 00 65 00 42 00 41 00 53 00 49 00 43 00 0D

Text read from sample.txt:
FreeBASIC

```

Platform Differences

- Unicode (w)strings are not supported in the DOS port of FreeB

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- QB had no support for Unicode

See also

- [Open](#)

End (Block)



Indicates the end of a compound statement block.

Syntax

```
End { Sub | Function | If | Select | Type | Enum | Scope | With  
Namespace | Extern | Constructor | Destructor | Operator |  
Property }
```

Description

Used to indicate the end of the most recent code block.

The type of the block must be included in the command: one of **Sub**, **Function**, **If**, **Select**, **Type**, **Enum**, **Scope**, **With**, **Namespace**, **Extern**, **Constructor**, **Destructor**, **Operator**, Or **Property**.

Ending a **Sub**, **Function**, **If**, **Select**, **Scope**, **Constructor**, **Destructor**, **Operator**, or **Property** block also closes the scope for variables defined inside that block. When the scope is closed, variables defined inside the scope are destroyed, calling their destructors as needed.

To end a program, see **End (Statement)**.

Example

```
Declare Sub checkvalue( n As Integer )  
  
Dim variable As Integer  
  
Input "Give me a number: ", variable  
If variable = 1 Then  
Print "You gave me a 1"  
Else  
Print "You gave me a big number!"  
End If  
checkvalue(variable)
```

```
Sub checkvalue( n As Integer )  
Print "Value is: " & n  
End Sub
```

Differences from QB

- none

See also

- **Constructor**
- **Destructor**
- **End (Statement)**
- **Enum**
- **Extern**
- **Function**
- **If...Then**
- **Namespace**
- **Operator**
- **Property**
- **Scope**
- **Select Case**
- **Sub**
- **Type**
- **With**

End (Statement)



Control flow statement to end the program.

Syntax

```
Declare Sub End ( ByVal retval As Long = 0 )
```

Usage

```
End [ retval ]
```

Parameters

retval

Error code returned to system.

Description

Used to exit the program, and return to the operating system. An option can be specified to indicate an error code to the system. If no return value is specified, the error code is automatically returned at the end of the program.

Usage of this statement does not cleanly close scope. Local variables are not called automatically, because FreeBASIC does not do stack unwinding. Global variables will be called in this case.

For this reason, it is discouraged to use **End** simply to mark the end of a program. It is better to come to an end automatically, and in a cleaner fashion, when the last statement is executed.

Example

```
' ' This program requests a string from the user, and returns an error code to the OS if the string was empty
Function main() As Integer
    ' assign input to text string
```

```

Dim As String text
Line Input "Enter some text ( try ""abc"" ): "

'' If string is empty, print an error message
'' return error code 1 (failure)
If( text = "" ) Then
    Print "ERROR: string was empty"
    Return 1
End If

'' string is not empty, so print the string and
'' return error code 0 (success)
Print "You entered: " & text
Return 0

End Function

'' call main() and return the error code to the OS
End main()

```

Differences from QB

- The END statement supports specifying a custom return value system.

See also

- End (Block)
- Return

Control flow statement for conditional branching.

Syntax

```
If expression Then : statement(s) [Else statement(s)] : End If  
or  
If expression Then  
statement(s)  
End If
```

Differences from QB

- None

See also

- [If...Then](#)

Declares an enumerated type.

Syntax

```
Enum [typename [ Explicit ] ]  
  symbolname [= expression] [, ...]  
  ...  
End Enum
```

Parameters

typename

Name of the **Enum**

symbolname

Name of the constant

expression

A constant expression

Explicit

Requires that symbols must be explicitly referred to by

typename.symbolname

Description

Enum, short for enumeration, declares a list of symbol names that correspond to discrete values. If no initial value is given, the first item will be set to 0. Each subsequent symbol has a value one more than the previous unless *expression* is given.

Symbols may be each on their own line, or separated on a single line by commas.

An **Enum** is a useful way of grouping together a set of related **constants**. A symbol can be accessed like a constant, e.g: `a = symbolname`. But if the name clashes with another symbol, it must be resolved using `typename.symbolname`. This resolution method is always required if you make the enum **Explicit**.

A non-**Explicit Enum** declared inside an **Extern ... End Extern** block will add its constants to the parent namespace directly, as in C, instead

of acting as a namespace on its own. It disallows the *typename.symbolname* style of access, and the constants may conflict with other symbols from the parent namespace.

An **Enum** can be passed as a user defined type to **overloaded** operator functions.

Example

```
Enum MyEnum
  option1 = 1
  option2
  option3
End Enum

Dim MyVar As MyEnum

MyVar = option1

Select Case MyVar
  Case option1
    Print "Option 1"
  Case option2
    Print "Option 2"
  Case option3
    Print "Option 3"
End Select
```

Dialect Differences

- Explicit Enum not available in the *-lang qb* dialect unless referenced with the alias `__Explicit`.

Differences from QB

- [New to FreeBASIC](#)

See also

- [Const](#)
- [Operator](#)

SetEnviron



Sets a system environment variable

Syntax

```
Declare Function SetEnviron ( ByRef varexpression As String ) As
```

Usage

```
result = SetEnviron( varexpression )
```

Parameters

varexpression

Name and setting of an environment variable in the following (or equivalent) format: `varname=varstring` (varname being the name of the environment variable, and varstring being the value to set it to).

Return Value

Return zero (0) if successful, non-zero otherwise.

Description

Modifies system environment variables. There are several variables a user can set, and some are default ones on your system. An example of this would be `fbgfx`, where `fbgfx` is the graphics driver the FreeBASIC graphics library will use.

Example

```
'e.g. to set the system variable "path" to "c:":  
Shell "set path" 'shows the value of path  
SetEnviron "path=c:"  
Shell "set path" 'shows the new value of path
```

```
' ' WINDOWS ONLY EXAMPLE! - We just set the graph
```

```

'' GDI rather than DirectX.
'' You may note a difference in FPS.
SetEnviron("fbgfx=GDI")

'' Desktop width/height
Dim As Integer ScrW, ScrH, BPP
ScreenInfo ScrW, ScrH, BPP

'' Create a screen at the width/height of your m
'' Normally this would be slow, but GDI is fairl
'' of thing.
ScreenRes ScrW, ScrH, BPP

'' Start our timer/
Dim As Double T = Timer

'' Lock our page
ScreenLock
Do

'' Print time since last frame
Locate 1, 1
Print "FPS: " & 1 / ( Timer - T )
T = Timer

'' Flip our screen
ScreenUnlock
ScreenLock
'' Commit a graphical change to our screen.
Cls

Loop Until Len(Inkey)

'' unlock our page.
ScreenUnlock

```

Differences from QB

- In QB, `setEnviron` was called `Environ`.

See also

- `Environ`
- `Shell`

Returns the value of a system environment variable

Syntax

```
Declare Function Environ ( ByRef varname As Const String ) As String
```

Usage

```
result = Environ( varname )
```

Parameters

varname

The name of an environment variable.

Return Value

Returns the text value of the environmental variable, or the empty string ("") if the variable does not exist.

Description

`Environ` returns the text value of a system environment variable.

Example

```
'e.g. to show the system variable "path":  
Print Environ("path")
```

Differences from QB

- The `Environ` statement is now called `SetEnviron`.

See also

- `SetEnviron`
- `Shell`

Checks to see if the end of an open file has been reached

Syntax

```
Declare Function EOF ( ByVal filenum As Long ) As Long
```

Usage

```
result = EOF( filenum )
```

Parameters

filenum

File number of an open file.

Return Value

Returns true (-1) if end-of-file has been reached, zero (0) otherwise.

Description

When reading from files opened for **Input (File Mode)**, it is useful to avoid avoiding errors caused by reading past the ends of files. Use EOF to check for the end of an already opened file. Use **FreeFile** to retrieve an available file file number.

For file numbers bound to files opened for **output**, EOF always returns false.

Example

```
' ' This code finds a free file number to use and a  
' "file.ext" and if successful, binds our file number  
' reads the file line by line, outputting it to the console  
' returns true, in this case we ignore the loop if it returns true
```

```
Dim As String file_name  
Dim As Integer file_num
```

```

file_name = "file.ext"
file_num = FreeFile( )           '' retrieve

'' open our file and bind our file number to it, e
If( Open( file_name For Input As #file_num ) ) Then
    Print "ERROR: opening file " ; file_name
    End -1
End If

Do Until EOF( file_num )        '' loop unt
    Dim As String text
    Line Input #file_num, text  '' rea
    Print text                   '' ... a
Loop

Close #file_num                 '' close fi

End 0

```

Because of underlying differences in the libraries used by the compiler when reading text files created in Linux (with LF line endings) in the Visual Basic compilers do not have this problem. One solution is to open the file for binary mode, which can still be used as in the above example, and the EOF function will return true after the last character, but this is only used to mark the end of text streams that are

Differences from QB

- In QB the comm port signaled an EOF when there were no characters to read
- In QB, for files opened in RANDOM or BINARY mode, EOF returned true after the last character had been attempted. In FreeBASIC, EOF returns true after the last character is read.

See also

- LOF
- LOC
- FreeFile

Operator Eqv (Equivalence)



Returns the bitwise-and (equivalence) of two numeric values

Syntax

```
Declare Operator Eqv ( ByRef lhs As T1, ByRef rhs As T2 ) As Ret
```

Usage

```
result = lhs Eqv rhs
```

Parameters

lhs

The left-hand side expression.

T1

Any numeric or boolean type.

rhs

The right-hand side expression.

T2

Any numeric or boolean type.

Ret

A numeric or boolean type (varies with *T1* and *T2*).

Return Value

Returns the bitwise-equivalence of the two operands.

Description

This operator returns the bitwise-equivalence of its operands, a logical operation that results in a value with bits set depending on the bits of the operands (for conversion of a boolean to an integer, false or true boolean value becomes 0 or -1 integer value).

The truth table below demonstrates all combinations of a boolean-equivalence operation:

Lhs Bit	Rhs Bit	Result

0	0	1
1	0	0
0	1	0
1	1	1

No short-circuiting is performed - both expressions are always evaluated.

The return type depends on the types of values passed. **Byte**, **UByte** and floating-point type values are first converted to **Integer**. If the left and right-hand side types differ only in signedness, then the return type is the same as the left-hand side type (τ_1), otherwise, the larger of the two types is returned. Only if the left and right-hand side types are both **Boolean**, the return type is also **Boolean**.

This operator can be overloaded for user-defined types.

Example

```
Dim As UByte a = &b00110011
Dim As UByte b = &b01010101, c
c = a Eqv b ' c = &b10011001
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- None

See also

- **Operator Truth Tables**

Erase



Statement to erase arrays

Syntax

```
Declare Sub Erase ( array As Any [, ... ] )
```

Usage

```
Erase( array0 [, array1 ... arrayN ] )
```

Parameters

array

An array to be erased.

Description

Using `Erase` on a fixed-length array clears (re-initializes) all elements.

Using `Erase` on a variable-length array (array already sized) frees the memory used by the element data (does not allow to after resize it wit a different number of dimensions).

Example

```
Dim MyArray1(1 To 10) As Integer
ReDim MyArray2(1 To 10) As Integer

Erase MyArray1, MyArray2
```

Differences from QB

- None

See also

- **Common**
- **Dim**
- **Extern**
- **LBound**
- **ReDim**
- **Static**
- **UBound**
- **Var**

Error reporting function

Syntax

```
Declare Function Erfn ( ) As ZString Ptr
```

Usage

```
result = Erfn ( )
```

Return Value

Returns a pointer to the string identifying the function where the error occurred.

Returns NULL if the source is not compiled with the **-exx** compiler option.

Description

An error reporting function returning a pointer to the name of the function.

Example

```
' test.bas
' compile with fbc -exx -lang fblite test.bas

#lang "fblite"

Sub Generate_Error
  On Error Goto Handler
  Error 1000
Exit Sub
Handler:
  Print "Error Function: "; *Erfn()
  Print "Error Module  : "; *Ermn()
```

```
Resume Next
End Sub

Generate_Error
```

Output:

```
Error Function: GENERATE_ERROR
Error Module  : test.bas
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__Erfn`.

Differences from QB

- New to FreeBASIC

See also

- [Er1](#)
- [Ermn](#)
- [On...Error](#)

Error handling function to return the line where the error occurred

Syntax

Declare Function **Er1** () As Integer

Usage

result = **Er1**

Return Value

An **Integer** return value containing the line number where the last error occurred.

Description

Er1 will return the line number where the last error occurred. If no error has occurred, **Er1** will return 0.

Er1 cannot always be used effectively -- QB-like error handling must be enabled.

Er1 is reset by RESUME and RESUME NEXT

Example

```
' compile with -lang fblite or qb
#lang "fblite"

' note: compilation with '-
ex' option is required

On Error Goto ErrorHandler
```

```
' Generate an explicit error
Error 100

End

ErrorHandler:
  Dim num As Integer = Err
  Print "Error "; num; " on line "; Err
  Resume Next

' Expected output is
' Error 100 on line 6
```

Differences from QB

- FreeBASIC returns the source code line number and ignores the values of all explicit line numbers, where as QB returns the last encountered explicit line number, and will return zero (0) when explicit line numbers are not used.

See also

- **Error Handling**
- **Err**

Error reporting function

Syntax

```
Declare Function Ernm ( ) As ZString Ptr
```

Usage

```
result = Ernm ( )
```

Return Value

Returns a pointer to the string identifying the module where the error occurred.

Returns NULL if the source is not compiled with the **-exx** compiler option.

Description

An error reporting function returning a pointer to the name of the module.

Example

```
' ' test.bas
' ' compile with fbc -exx -lang fblite test.bas

#lang "fblite"

Sub Generate_Error
  On Error Goto Handler
  Error 1000
  Exit Sub
Handler:
  Print "Error Function: "; *ErFn()
  Print "Error Module   : "; *ErMn()
```

```
Resume Next
End Sub

Generate_Error
```

Output:

```
Error Function: GENERATE_ERROR
Error Module  : test.bas
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__Ernm`.

Differences from QB

- New to FreeBASIC

See also

- [Erfn](#)
- [Er1](#)
- [On...Error](#)

Get or set the run-time error number

Usage

```
result = Err( )  
or  
Err = number
```

Description

The **Err()** function returns the FreeBASIC run-time error number which is set by the built-in statements and functions, or by the program through **number** or **Error**. Unlike **Error**, **Err = number** sets the error number without invoking an error handler.

See [Runtime Error Codes](#) for a listing of the predefined runtime error codes and their associated meaning. The program may use additional custom error numbers.

Err can always be used, even if QB-like error handling is not enabled. It is reset by [Resume](#) and [Resume Next](#).

Note: Care should be taken when calling an internal function (such as `Print`) after an error occurred, because it will reset the error value with its own status. To preserve the **Err** value, it is a good idea to store it in a variable as soon as the error handler is entered.

Example

An example using QBasic style error handling (compile with `-ex` option)

```
' ' Compile with -lang fblite or qb  
  
#lang "fblite"  
  
On Error Goto Error_Handler  
Error 150  
End
```

```
Error_Handler:
  n = Err()
  Print "Error #"; n
  Resume Next
```

An example using inline error handling (note: `Open` can also return its status when called as a function)

```
' ' compile without -e switch

Dim filename As String

Do
  Line Input "Input filename: ", filename
  If filename = "" Then End
  Open filename For Input As #1
Loop Until Err() = 0

Print Using "File '&' opened successfully"; filename
Close #1
```

Differences from QB

- Error numbers are not the same as in QB.

See also

- **On Error**
- **Error**
- **Error Handling**
- **Runtime Error Codes**

Error handling statement to force an error to be generated

Syntax

```
Declare Sub Error ( errno As Integer )
```

Usage

Error *number*

Parameters

number

The error number to generate

Description

Error invokes the error handler specified with **On Error** or, in case there was none set, aborts the program, printing an error message similar to those generated by the compiler's -exx run-time error checking. It's possible to use the **built-in run-time error numbers** and/or other custom error numbers for *number*. This can be used to simulate custom error numbers.

Example

To send an error alert of error 150 (just some arbitrary error code) one would do the following:

```
Error 150
```

Differences from QB

- Error numbers are not the same as in QB.

See also

- **Err**
- **Error Handling**
- **Runtime Error Codes**

Event (Message Data From Screenshot)



Pre-defined structure (UDT) from fbgfx.bi used by **ScreenEvent** to return event data

Syntax

```
#include once "fbgfx.bi"  
using fb  
Dim variable As Event
```

Description

Here we report the EVENT structure for clarity:

```
Type EVENT Field = 1  
  Type As Long  
  Union  
    Type  
      scancode As Long  
      ascii As Long  
    End Type  
    Type  
      x As Long  
      y As Long  
      dx As Long  
      dy As Long  
    End Type  
  button As Long  
  z As Long  
  w As Long  
  End Union  
End Type
```

The `Type` field will contain the event type ID, while the remaining 4 integers will hold sensitive data to the event type.

Event types

The event type is identified by an ID number returned into the first integer of the *event* buffer (the `.type` field in the `EVENT` structure).

Known event type IDs - and their values at time of writing - are:

- `EVENT_KEY_PRESS` (1) A key was pressed on the keyboard. The `.scancode` field contains the platform independent scancode value for the key; if the key has an ascii representation, it is held into the `.ascii` field, which otherwise has a value of 0.
- `EVENT_KEY_RELEASE` (2) A key was released on the keyboard. The `.scancode` and `.ascii` fields have the same meaning as with the `EVENT_KEY_PRESS` event.
- `EVENT_KEY_REPEAT` (3) A key is being held down repeatedly. The `.scancode` and `.ascii` fields have the same meaning as with the `EVENT_KEY_PRESS` event.
- `EVENT_MOUSE_MOVE` (4) The mouse was moved while it was on the program window. The `.x` and `.y` fields contain the new mouse position relative to the upper-left corner of the screen, while the `.dx` and `.dy` fields contain the motion deltas.
- `EVENT_MOUSE_BUTTON_PRESS` (5) One of the mouse buttons was pressed. The `.button` field has one bit set identifying the button that was pressed; bit 0 identifies the left mouse button, bit 1 the right mouse button and bit 2 the middle mouse button.
- `EVENT_MOUSE_BUTTON_RELEASE` (6) One of the mouse buttons was released. The `.button` field has the same meaning as with the `EVENT_MOUSE_BUTTON_PRESS` event.
- `EVENT_MOUSE_DOUBLE_CLICK` (7) One of the mouse buttons was double clicked. The `.button` field has the same meaning as with the `EVENT_MOUSE_BUTTON_PRESS` event.
- `EVENT_MOUSE_WHEEL` (8) The mouse wheel was used; the new wheel position is returned into the `.z` field.
- `EVENT_MOUSE_ENTER` (9) The mouse was moved into the program window.
- `EVENT_MOUSE_EXIT` (10) The mouse was moved out of the

program window.

- `EVENT_WINDOW_GOT_FOCUS` (11) The program window has got focus.
- `EVENT_WINDOW_LOST_FOCUS` (12) The program window has lost focus.
- `EVENT_WINDOW_CLOSE` (13) The user attempted to close the program window.
- `EVENT_MOUSE_HWHEEL` (14) The horizontal mouse wheel was used; the new horizontal wheel position is returned into the `.w` field.

The `fbgfx.bi` header file contains a definition of the `EVENT` user data type, so it is not necessary to declare it manually.

Dialect Differences

- In *lang fb*, the structure and constants are stored in the `FB Namespace`. This is not the case in other dialects.

Differences from QB

- New to FreeBASIC

See also

- [ScreenEvent](#)

Temporarily transfers execution to an external program

Syntax

```
Declare Function Exec ( ByRef program As Const String, ByRef arg  
String ) As Long
```

Usage

```
result = Exec( program, arguments )
```

Parameters

program

The file name (including file path) of the program (executable) to trans
arguments

The command-line arguments to be passed to the program.

Return Value

The exit status of the program, or negative one (-1) if the program cou

Description

Transfers control over to an external program. When the program exit resumes immediately after the call to **Exec**.

Example

```
'A Windows based example but the same idea applies  
Const exename = "NoSuchProgram.exe"  
Const cmdline = "arg1 arg2 arg3"  
Dim result As Integer  
result = Exec( exename, cmdline )  
If result = -1 Then  
    Print "Error running "; exename  
Else  
    Print "Exit code: "; result
```

End If

Platform Differences

- Linux requires the *program* case matches the real name of the DOS are case insensitive. The program being executed may b its command line parameters.
- Path separators in Linux are forward slashes / . Windows uses but it allows for forward slashes . DOS uses backward \ slashe
- Exit code is limited to 8 bits in DOS.

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- **Chain** transfer temporarily, without arguments
- **Run** one-way transfer
- **Command** pick arguments

ExePath



Returns the path of the running program

Syntax

```
Declare Function ExePath ( ) As String
```

Usage

```
result = ExePath
```

Return Value

A **String** variable set to the path of the running program.

Description

Returns the path (the location) of the calling program. This is not necessarily the same as **CurDir**.

Example

```
Dim pathname As String = ExePath
Print "This program's initial directory is: " & pa
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the `__Exepath`.

Differences from QB

- New to FreeBASIC

See also

- CurDir

Control flow statement to exit a compound statement block

Syntax

```
Exit {Do | For | While | Select }  
Exit {Sub | Function | Operator | Property }
```

```
Exit {Do [, Do [ , ...] ] |  
For [, For [ , ...] ] |  
While [, While, [...] ] |  
Select [, Select [ , ...] ] }
```

Description

Leaves a code block such as a **Sub**, **Function**, **Do...Loop**, **For...Next**, **While...Wend**, **Function**, **Operator**, **Property**. The execution skips the rest of the block and goes to the line after its end.

Where there are multiple **Do / For / While / Select** blocks nested, it will exit the innermost block of that type. You can skip to the end of multiple blocks of that type by separating them by commas.

For example: `Exit while, while`

Example

```
'e.g. the print command will not be seen
```

```
Do  
    Exit Do ' Exit the Do...Loop and continues to  
    Print "I will never be shown"  
Loop
```

```
Dim As Integer i, j  
For i = 1 To 10
```

```
For j = 1 To 10
    Exit For, For
Next j
Print "I will never be shown"
Next i
```

Differences from QB

- EXIT WHILE and EXIT SELECT are new to FreeBASIC.

See also

- Sub
- Function
- Do...Loop
- For...Next
- While...Wend
- Continue

Returns e raised to the power of a given number

Syntax

```
Declare Function Exp cdecl ( ByVal number As Double ) As Double
```

Usage

```
result = Exp( number )
```

Parameters

number

The **Double** *number* that e is raised to the power of.

Return Value

Returns the **Double** value of e raised to power of *number*.

Description

The mathematical constant e , also called Euler's constant, is the base significant figures is: 2.7182818284590452354. The required *number* argument **Exp** returns infinity. If *number* is too small, **Exp** returns zero (0.0). If *number*

Example

```
'Compute Continuous Compound Interest
Dim r As Double
Dim p As Double
Dim t As Double
Dim a As Double

Input "Please enter the initial investment (princi
Input "Please enter the annual interest rate (as a
Input "Please enter the number of years to invest:
```

```
a = p * Exp ( r * t )  
Print ""  
Print "After";t;" years, at an interest rate of";
```

The output would look like:

```
Please enter the initial investment (principal amount): 100  
Please enter the annual interest rate (As a decimal): .08  
Please enter the number of years To invest: 20  
After 20 years, at an interest rate of 8%, your initial invest
```

Differences from QB

- None

See also

- [Log](#)
- [Operator ^ \(Exponentiate\)](#)

Declaration specifier to indicate that a procedure in a DLL should be visible from other programs

Syntax

```
{ Sub | Function } proc_name ( argumentlist ) [ As datatype ]  
Export
```

Description

If a function is declared with this clause in a DLL, it is added to the public export table, so external programs can dynamically link to it using [DyLibSymbol](#).

Example

See the examples on the [Shared Libraries](#) page.

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__Export`.

Platform Differences

- Dynamic link libraries are not available in DOS, as the OS doesn't support them.

Differences from QB

- New to Freebasic

See also

- [DyLibLoad](#)
- [DyLibSymbol](#)

- **Alias**

Specifies a base type from which to derive a new type

Syntax

```
Type|Union typename Extends base_typename
...
End Type|Union
```

Description

Extends declares *typename* to be derived from *base_typename*. The derived type inherits all members of the base type. *typename* objects may be used in place of *base_typename* objects. Regular members of *typename* are like regular members of *base_typename*.

However, a regular member will shadow an inherited member if they both exist. To explicitly access members of the base type shadowed by local members, use the **Base** keyword.

User-defined types that extend another type will include the base type's size plus the size needed for any regular members. If the base type is not required to have regular members of its own.

In *typename* (the derived user-defined type), the fields can share the size of the base type. It does not matter whether *base_typename* is a **Union** or not.

If only *base_typename* is a **Union**, then it will not be affected by fields from *typename*. As a **Union** is not allowed to have complex fields (i.e. user-defined types), *typename* is allowed to have (contain) a complex *base_typename*.

The **Base (Initializer)** keyword can be used at the top of constructors for the base type.

Extending the built-in **Object** type allows a user-defined type to be used with **Abstract** methods, and to use the **Override** method attribute.

Note: Derived UDT pointer can only be casted to "compatible" pointer first.

Warning: Before fbc version 0.24, these five keywords dedicated to inheritance (**Virtual**, **Abstract**, **Override**, **Base**, **Operator Is**) were not supported. Three new keywords **Virtual**, **Abstract**, and **Operator Is** were added.

Example

```
Type SchoolMember 'Represents any school member'
  Declare Constructor ()
  Declare Sub Init (ByRef _name As String, ByVal
  As String Name
  As Integer age
End Type

Constructor SchoolMember ()
  Print "Initialized SchoolMember"
End Constructor

Sub SchoolMember.Init (ByRef _name As String, ByVal
  This.name = _name
  This.age = _age
  Print "Name: "; This.name; "   Age:"; This.age
End Sub

Type Teacher Extends SchoolMember 'Represents a te
  Declare Constructor (ByRef _name As String, By
  As Integer salary
  Declare Sub Tell ()
End Type

Constructor Teacher (ByRef _name As String, ByVal
  Print "Initialized Teacher"
  This.Init(_name, _age) 'implicit access to bas
  This.salary = _salary
End Constructor

Sub Teacher.Tell ()
  Print "Salary:"; This.salary
End Sub
```

```

Type Student Extends SchoolMember 'Represents a student
    Declare Constructor (ByRef _name As String, ByVal _age As Integer, ByVal _marks As Integer marks)
    Declare Sub Tell ()
End Type

Constructor Student (ByRef _name As String, ByVal _age As Integer, ByVal _marks As Integer marks)
    Print "Initialized Student"
    This.Init(_name, _age) 'implicit access to base class constructor
    This.marks = _marks
End Constructor

Sub Student.Tell ()
    Print "Marks:"; This.marks
End Sub

Dim As Teacher t = Teacher("Mrs. Shrividya", 40, 30)
t.Tell()
Print "Teacher Name: " & t.name
Dim As Student s = Student("Swaroop", 22, 75)
s.Tell()

```

```

' Example using all eight keywords of inheritance:
' 'Extends', 'Base.', 'Base()', 'Object', 'Is' class
Type root Extends Object ' 'Extends' to activate Reflection
    Declare Function ObjectHierarchy () As String
    Declare Abstract Function ObjectRealType () As String

    Dim Name As String
    Declare Virtual Destructor () ' 'Virtual' declaration
Protected:
    Declare Constructor () ' to avoid user constructor
    Declare Constructor (ByRef rhs As root) ' ' to avoid user constructor
End Type ' derived type may be member data empty

```

```

Constructor root ()
End Constructor

Function root.ObjectHierarchy () As String
    Return "Object(forRTTI) <- root"
End Function

Virtual Destructor root ()
    Print "root destructor"
End Destructor

Type animal Extends root ' 'Extends' to inherit of
    Declare Constructor (ByRef _name As String = "")
    Declare Function ObjectHierarchy () As String
    Declare Virtual Function ObjectRealType () As String

    Declare virtual Destructor () Override ' 'Virtual
    ' 'Override

End Type

Constructor animal (ByRef _name As String = "")
    This.name = _name
End Constructor

Function animal.ObjectHierarchy () As String
    Return Base.ObjectHierarchy & " <- animal" ' 'Base
End Function

Virtual Function animal.ObjectRealType () As String
    Return "animal"
End Function

Virtual Destructor animal ()
    Print " animal destructor: " & This.name
End Destructor

```

```

Type dog Extends animal ' 'Extends' to inherit of
  Declare Constructor (ByRef _name As String = "")
  Declare Function ObjectHierarchy () As String
  Declare Function ObjectRealType () As String Over

  Declare Destructor () Override ' 'Override' to c
End Type ' derived type may be member data empty

Constructor dog (ByRef _name As String = "")
  Base(_name) ' 'Base()' allows to call parent cor
End Constructor

Function dog.ObjectHierarchy () As String
  Return Base.ObjectHierarchy & " <- dog" ' 'Base.
End Function

Function dog.ObjectRealType () As String
  Return "dog"
End Function

Destructor dog ()
  Print "   dog destructor: " & This.name
End Destructor

Type cat Extends animal ' 'Extends' to inherit of
  Declare Constructor (ByRef _name As String = "")
  Declare Function ObjectHierarchy () As String
  Declare Function ObjectRealType () As String Over

  Declare Destructor () Override ' 'Override' to c
End Type ' derived type may be member data empty

Constructor cat (ByRef _name As String = "")
  Base(_name) ' 'Base()' allows to call parent cor
End Constructor

Function cat.ObjectHierarchy () As String

```

```

Return Base.ObjectHierarchy & " <- cat" 'Base.
End Function

Function cat.ObjectRealType () As String
Return "cat"
End Function

Destructor cat ()
Print "    cat destructor: " & This.name
End Destructor

Sub PrintInfo (ByVal p As root Ptr) ' must be put
Print " " & p->Name, " " & p->ObjectRealType,
If *p Is dog Then ' 'Is' allows to check compati
Print Cast(dog Ptr, p)->ObjectHierarchy
ElseIf *p Is cat Then ' 'Is' allows to check com
Print Cast(cat Ptr, p)->ObjectHierarchy
ElseIf *p Is animal Then ' 'Is' allows to check
Print Cast(animal Ptr, p)->ObjectHierarchy
End If
End Sub

Print "Name:", "Object (real):          Hierarchy:"
Dim a As root Ptr = New animal("Mouse")
PrintInfo(a)
Dim d As root Ptr = New dog("Buddy")
PrintInfo(d)
Dim c As root Ptr = New cat("Tiger")
PrintInfo(c)
Print
Delete a
Delete d
Delete c

```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- **Type**
- **Union**
- **Base (Initializer)**
- **Base (Member Access)**
- **Object**
- **Operator Is**
- **Virtual**
- **Abstract**
- **Override**

Extern



Declares a variable, array or object having external linkage

Syntax

```
Extern [ Import ] symbolName[ (subscripts) ] [ Alias "aliasname"  
] As DataType [, ...]
```

Parameters

symbolName

The name of the variable, array or object.

aliasname

An alternate external name for the variable, array or object.

Description

Declares *symbolName* as an external name, meaning it is global to external modules. **Extern** only declares variables, arrays and objects, and does not define them (different from **Common** or **Dim**). It also has the effect of making *symbolName* a *shared* name, meaning it is visible within procedures (see **Shared**). A *symbolName* declared as external name can be (re)defined (using **Dim** or **Redim**) only in a single external module.

If **Alias** is used, *aliasname* will be used as the external name rather than *symbolName*, and its case will be preserved.

If **Import** is used, the name will be added to the dynamic library import list so its address can be fixed at run-time.

Example

```
' ' extern1.bas  
  
Extern Foo Alias "foo" As Integer  
  
Sub SetFoo  
    foo = 1234
```

```
End Sub
```

```
' ' extern2.bas  
  
Declare Sub SetFoo  
  
Extern Foo Alias "foo" As Integer  
  
Dim foo As Integer = 0  
  
SetFoo  
  
Print Foo
```

Output:

```
1234
```

Dialect Differences

- Not available in the *-lang qb* dialect.

Differences from QB

- New to FreeBASIC

See also

- [Extern...End Extern](#)
- [Common](#)
- [Dim](#)
- [Shared](#)

Extern...End Extern



Statement block to allow calling of functions compiled for specific language

Syntax

```
Extern { "C" | "C++" | "Windows" | "Windows-MS" } [ Lib "libname"
  declarative statements
End Extern
```

Description

Extern blocks provide default calling conventions for procedures and functions

Extern "c" blocks provide a default **cdecl** calling convention to procedures declared within them. The same effect can be achieved without the **Extern** block by using an **Alias** string containing the exact procedure name.

Extern "c++" blocks are exactly like **Extern "c"** blocks but they also make the calling convention compatible to that of *g++-4.x*.

Extern "Windows" blocks provide a default **stdcall** calling convention to procedures declared within them, and on the Windows platform, append an "@N" suffix to the procedure name, where N is the size in bytes of any procedure parameters. Similar to the **Extern "c"** block, you can use **stdcall** and **Alias**.

Extern "Windows-MS" blocks are exactly like **Extern "Windows"** blocks but they use the MS naming convention on Windows.

Lib "libname" can be used to specify a library which will be linked in and used. Additionally, all procedure declarations inside the **Extern** block must be specified as part of their declarations (but it can still be overridden with

Example

```
' ' This procedure uses the default calling convention
' ' STDCALL on Win32 and CDECL on Linux/DOS/*BSD, and
' ' "MYTEST1@4" on Win32 and "MYTEST1" on Linux/DOS
' ' ALL-UPPER-CASE name mangling).
```

```

Sub MyTest1 ( ByVal i As Integer )
End Sub

Extern "C"
    ' This procedure uses the CDECL convention ar
    ' as "MyTest2".
    Sub MyTest2 ( ByVal i As Integer )
    End Sub
End Extern

Extern "C++"
    ' This procedure uses the CDECL convention ar
    ' compatible to g++-4.x, specifically: "_Z7My
    Sub MyTest3 ( ByVal i As Integer )
    End Sub
End Extern

Extern "Windows"
    ' This procedure uses the STDCALL convention
    ' as "MyTest4@4" on Windows, and "MyTest4" or
    Sub MyTest4 ( ByVal i As Integer )
    End Sub
End Extern

Extern "Windows-MS"
    ' This procedure uses the STDCALL convention
    ' as "MyTest5".
    Sub MyTest5 ( ByVal i As Integer )
    End Sub
End Extern

MyTest1( 0 )
MyTest2( 0 )
MyTest3( 0 )
MyTest4( 0 )

```

Dialect Differences

- Extern blocks are only available in the *-lang fb* dialect.

Differences from QB

- New to FreeBASIC

Platform Differences

- On Linux, *BSD and DOS platforms, Extern "Windows" blocks I and thus are equal to Extern "Windows-MS".

See also

- [cdecl](#)
- [stdcall](#)
- [Extern](#)

False



Intrinsic constant set by the compiler

Syntax

```
Const False As Boolean
```

Description

Gives the False `Boolean` value where used.

Example

```
Dim b As Boolean = False
If b Then
    Print "b is True"
Else
    Print "b is False"
End If
```

```
b is False
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__False`.

Differences from QB

- New to FreeBASIC

See also

- **True**
- **Boolean**

Field



Specifies field alignment.

Syntax

```
Type|Union typename Field = { 1 | 2 | 4 }  
...  
End Type|Union
```

Description

Field can be used to pack **Types** or **Unions** more tightly than **the default**. The most commonly used value is **Field = 1**, which causes the **Type** to be packed as tightly as possible, without any padding bytes being added between the fields or at the end of the **Type**. **Field** can only be used to control field alignment, but it cannot be used to increase it. In order to add padding bytes, a **Union** with appropriate members could be used instead.

Example

```
Type bitmap_header Field = 1  
  bfType           As UShort  
  bftype           As UInteger  
  bfReserved1     As UShort  
  bfReserved2     As UShort  
  bfOffBits       As UInteger  
  biSize           As UInteger  
  biWidth          As UInteger  
  biHeight        As UInteger  
  biPlanes        As UShort  
  biBitCount      As UShort  
  biCompression   As UInteger  
  biSizeImage     As UInteger  
  biXPelsPerMeter As UInteger  
  biYPelsPerMeter As UInteger  
  biClrUsed       As UInteger  
  biClrImportant  As UInteger  
End Type
```

```
Dim bmp_header As bitmap_header

'Open up bmp.bmp and get its header data:
'Note: Will not work without a bmp.bmp to load . .
Open "bmp.bmp" For Binary As #1

    Get #1, , bmp_header

Close #1

Print bmp_header.biWidth, bmp_header.biHeight

Sleep
```

Dialect Differences

- In the *-lang qb* dialect, the compiler assumes `Field = 1` by default if no other `Field` was specified, causing all structures to be tightly packed without added padding, as in QB.

Differences from QB

- In QB `Field` was used to define fields in a file buffer at run time. This feature is not implemented in FB, so the keyword has been replaced. To define fields in a file buffer, `Types` must be used.

See also

- [Type](#)
- [Union](#)
- [Structure packing/field alignment](#)

Returns information about an open file number

Syntax

```
Declare Function FileAttr ( ByVal filenum As Long, ByVal returntype As Long = 1 ) As Integer
```

Usage

```
#include "file.bi"  
result = FileAttr( filenum, [ returntype ] )
```

Or

```
#include "vbcompat.bi"  
result = FileAttr( filenum, [ returntype ] )
```

Parameters

filenum

The file number of a file or device opened with **open**

returntype

An integer value indicating the type of information to return.

Return Value

A value associated with the return type, otherwise 0 on error.

Description

Information about the file number is returned based on the supplied *returntype*

Value	Description	constant
1	File Mode	fbFileAttrMode
2	File Handle	fbFileAttrHandle
3	Encoding	fbFileAttrEncoding

For File Mode, *returntype* = 1 (fbFileAttrMode) the return value is the

one or more of the following values:

Value	File Mode	Constant
1	Input	fbFileModeInput
2	Output	fbFileModeOutput
4	Random	fbFileModeRandom
8	Append	fbFileModeAppend
32	Binary	fbFileModeBinary

For File Handle, *returntype* = 2 (fbFileAttrHandle), the return value is handle as supplied by the C Runtime for file-type devices.

On Windows only: For File Handle, *returntype* = 2 (fbFileAttrHandle) returned for COM devices is the handle returned by `createFile()` when the device was first opened. The value returned for LPT devices is the handle returned by `openPrinter()` when the device was first opened. This handle can be passed to other Windows API functions.

On Linux only: For File Handle, *returntype* = 2 (fbFileAttrHandle), the value returned for COM devices is the file descriptor returned by `open()` when the device was first opened.

For Encoding, *returntype* = 3 (fbFileAttrEncoding), the return value is one of the following values:

Value	Encoding	Constant
0	Ascii	fbFileEncodASCII
1	UTF-8	fbFileEncodUTF8
2	UTF-16	fbFileEncodUTF16
3	UTF-32	fbFileEncodUTF32

Example

```
#include "vbcompat.bi"  
#include "crt.bi"
```

```

Dim f As FILE Ptr, i As Integer

' Open a file and write some text to it

Open "test.txt" For Output As #1
f = Cast( FILE Ptr, FileAttr( 1, fbFileAttrHandle
For i = 1 To 10
    fprintf( f, !"Line %i\n", i )
Next i
Close #1

' re-open the file and read the text back

Open "test.txt" For Input As #1
f = Cast( FILE Ptr, FileAttr( 1, fbFileAttrHandle
While feof(f) = 0
    i = fgetc(f)
    Print Chr(i);
Wend
Close #1

```

Differences from QB

- None for *returntype* = 1
- QBasic and 16-bit Visual Basic returned DOS file handle for *re* 2
- *returntype* = 3 is new to FreeBASIC

See also

- [Open](#)

Copies a file

Syntax

```
Declare Function FileCopy ( ByVal source As ZString Ptr, ByVal  
destination As ZString Ptr ) As Long
```

Usage

```
#include "file.bi"  
FileCopy source, destination
```

or

```
#include "file.bi"  
result = FileCopy( source, destination )
```

Parameters

source

A **String** argument specifying the filename of the file to copy from. This file must exist.

destination

A **String** argument specifying the filename of the file to copy to. This file will be overwritten if it exists. This file should not be currently referenced by any open file handles.

Return Value

Returns 0 on success, or 1 if an error occurred.

Description

Copies the contents of the source file into the destination file, overwriting the destination file if it already exists.

It is necessary to **#include** either "file.bi" or "vbcompat.bi" in order to gain access to this function.

Example

```
#include "file.bi"  
FileCopy "source.txt", "destination.txt"
```

Platform Differences

- Linux requires the *filename* case matches the real name of the file. Windows and DOS are case insensitive.
- Path separators in Linux are forward slashes /. Windows uses backward slashes \ but it allows forward slashes. DOS uses backward slashes \.

Differences from QB

- New to FreeBASIC. Existed in Visual Basic.

See also

FileDateTime



Returns the last modified date and time of a file as **Date Serial**

Syntax

```
Declare Function FileDateTime ( ByVal filename As ZString Ptr )  
As Double
```

Usage

```
#include "file.bi"  
result = FileDateTime( filename )
```

or

```
#include "vbcompat.bi"  
result = FileDateTime( filename )
```

Parameters

filename
Filename to retrieve date and time for.

Return Value

Returns a **Date Serial**.

Description

Returns the file's last modified date and time as **Date Serial**.

Example

```
#include "vbcompat.bi"  
  
Dim filename As String, d As Double  
  
Print "Enter a filename: "  
Line Input filename
```

```
If FileExists( filename ) Then
    Print "File last modified: ";
    d = FileDateTime( filename )
    Print Format( d, "yyyy-mm-dd hh:mm AM/PM" )
Else
    Print "File not found"
End If
```

Platform Differences

- Linux requires the *filename* case matches the real name of the file. Windows and DOS are case insensitive.
- Path separators in Linux are forward slashes /. Windows uses backward slashes \ but it allows forward slashes. DOS uses backward slashes \.

Differences from QB

- New to FreeBASIC

See also

- **Date Serials**

Tests the existence of a file

Syntax

```
Declare Function FileExists ( ByVal filename As ZString Ptr ) As Long
```

Usage

```
#include "file.bi"  
result = FileExists( filename )
```

or

```
#include "vbcompat.bi"  
result = FileExists( filename )
```

Parameters

filename
Filename to test for existence.

Return Value

Returns non-zero (-1) if the file exists, otherwise returns zero (0).

Description

FileExists tests for the existence of a file. Internally, it may issue an `Open()` and a `Close()` function, which may have consequences - eg, any existing **Lock(s)** on the file may be released. Depending on the exact requirements, alternative methods of checking for file existence may be to use the `Dir()` function (being careful of attributes and ensuring the path doesn't contain wildcards), or to try **opening** the file and checking the return value for success.

Example

```
#include "vbcompat.bi"

Dim filename As String

Print "Enter a filename: "
Line Input filename

If FileExists( filename ) Then
    Print "File found: " & filename
Else
    Print "File not found: " & filename
End If
```

Platform Differences

- Linux requires the *filename* case matches the real name of the file. Windows and DOS are case insensitive.
- Path separators in Linux are forward slashes /. Windows uses backward slashes \ but it allows for forward slashes. DOS use backward \ slashes.

Differences from QB

- New to FreeBASIC

See also

- [Dir](#)

FileLen



Finds the length of a file given its filename

Syntax

```
Declare Function FileLen ( filename As String ) As LongInt
```

Usage

```
#include "file.bi"  
result = FileLen(filename)
```

or

```
#include "vbcompat.bi"  
result = FileLen(filename)
```

Parameters

filename

A **String** argument specifying the filename of the file whose length to return.

Description

Returns the size in bytes of the file specified by *filename*.

Example

```
#include "file.bi"  
Dim length As Integer  
length = FileLen("file.txt")
```

Platform Differences

- Linux requires the *filename* case matches the real name of the file. Windows and DOS are case insensitive.

- Path separators in Linux are forward slashes / . Windows uses backward slashes \ but it allows for forward slashes . DOS use backward \ slashes.

Differences from QB

- New to FreeBASIC. Existed in Visual Basic.

See also

- [LOF](#)

Returns the integer part of a number, rounding towards zero

Syntax

```
Declare Function Fix ( ByVal number As Single ) As Single
Declare Function Fix ( ByVal number As Double ) As Double
Declare Function Fix ( ByVal number As Integer ) As Integer
Declare Function Fix ( ByVal number As UInteger ) As UInteger
Declare Function Fix ( ByVal number As LongInt ) As LongInt
Declare Function Fix ( ByVal number As ULongInt ) As ULongInt
```

Usage

```
result = Fix( number )
```

Parameters

number
the floating-point number to truncate

Return Value

Returns the integer part of *number*, rounding towards zero.

Description

Equivalent to: `Sgn(number) * Int(Abs(number))`. For example, `Fix(1.3)` will return `1.0`, and `Fix(-4.9)` will return `-4.0`. For integer types, the number is returned unchanged.

Note: this function is also equivalent to `number - Frac(number)`.

The `Fix` unary **operator** can be overloaded with user defined types.

Example

```
Print Fix(1.9) ' will print 1
Print Fix(-1.9) ' will print -1
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- None

See also

- [Int](#)
- [Frac](#)
- [CInt](#)
- [Operator](#)

Flip



Changes the current video display page

Syntax

```
Declare Sub Flip ( ByVal frompage As Long = -1, ByVal topage As
```

Usage

```
Flip [ frompage ] [, topage ]
```

Parameters

frompage
previous page
topage
new page to display

Description

In normal graphics mode, **Flip** is an alias for **PCopy** and **ScreenCopy**. S

In OpenGL mode, **Flip** does a hardware page flip and displays the co
while in OpenGL mode, otherwise your app may also become unresp

Example

```
ScreenRes 320, 240, 32, 2      'Sets up the screen t

For n As Integer = 50 To 270

    ScreenSet 1,0      'Sets the working page to 1
    Cls
    Circle (n, 50),50 ,RGB(255,255,0) 'Draws a cir
    Flip 1,0          'Copies our circle from page 1 to

    Sleep 25
```

Next

```
Print "Now wasn't that neat!"  
Print "Push any key."  
Sleep
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

Control flow statement, open statement clause, or operator depending on context.

Syntax

```
For iterator = startvalue To endvalue [ Step increment ]  
or  
Open [ device ] "filename" For filemode As #handle  
or  
declare operator For ( byref stp as datatype )
```

See also

- **For...Next**
- **Open**
- **Operator**

Control flow statement for looping

Syntax

```
For iterator [ As datatype ] = startvalue To endvalue [ Step stepvalue ]  
  [ statement block ]  
Next [ iterator ]
```

Parameters

iterator

a variable identifier that is used to iterate from an initial value to an end value

If specified, the variable *iterator* will automatically be declared with the initial value

startvalue

an expression that denotes the starting value of the iterator

endvalue

an expression used to compare with the value of the iterator

stepvalue

an expression that is added to the iterator after every iteration

Description

A **For . . . Next** loop initializes *iterator* to *startvalue*, then executes the *statement block* by *stepvalue* until it exceeds *endvalue*. If *stepvalue* is not expressed, the default is 1.

The values of *stepvalue* and *endvalue* are stored internally immediately before the *statement block* and thus neither can be changed inside the **For** loop. Computations of *stepvalue* or *endvalue* will not be effective because the expressions used to define them are not evaluated while the loop is running. (The results of the expressions used to define them may be stored, but will not affect the execution of the **For** loop.) See examples.

Note: In some dialects, the temporary variables holding *stepvalue* and *endvalue* are reset at the end of the loop, and their values are not guaranteed to remain unchanged after the loop has been executed. For this reason, it is recommended never to use **Goto** (using **Goto** or similar), and then jump back into the middle of it later with a **deprecated** dialect.

The iterator must be an intrinsic scalar: only **Static/Shared** variables are allowed.

other kind can be used, including array elements, UDT members, **ByRef** dereferenced address.

The *iterator* may be defined having the same scope as the **For** state syntax. With this syntax, *iterator* is created and destroyed within the differences below.

If *endvalue* is less than *startvalue* then a negative *stepvalue* must be not execute at all, since *startvalue* compares greater than *endvalue*.

The **For** statement causes the execution of the statements in the *statement block* **greater than** *endvalue* (or **less than** *endvalue* if *stepvalue* < 0). *iterator* of *stepvalue* following each execution of the *statement block*. If an *iterator* is implicitly incremented by 1.

If an **Exit For** statement is encountered inside the *statement block*, the loop resumes immediately following the enclosing **Next** statement. If a **Continue For** statement is encountered, the rest of the *statement block* is skipped until the block's corresponding value is incremented and the loop restarted if it is still within the bounds.

Note: for integer data types, it is not possible to loop up to the highest possible value (or lowest possible value) that can be stored in the variable type, because the incremented variable exceeds *endvalue*, which can never happen. For example, for a variable from 0 to 255, the loop will only break once the variable reaches 256. For a variable for the counter wouldn't work, because although it can hold the value 256. See **Standard Data Type Limits** to find the upper and lower limits.

Like all control flow statements, the **For** statement can be nested, that is, inside a block of another **For** statement.

For, *Next*, and *Step* are operators that can be overloaded inside user code. See **Operator Next**, **Operator Step**.

Example

```
Print "counting from 3 to 0, with a step of -0.5"  
For i As Single = 3 To 0 Step -0.5  
    Print "i is " & i
```

```
Next i
```

```
Dim As Integer i, j, k, toTemp, stepTemp
j = 9: k = 1

For i = 0 To j Step k

    j = 0: k = 0 ' Changing j and k has no effect
    Print i;

Next i
Print

' Internally, this is what the above example does:
j = 9: k = 1

i = 0: toTemp = j: stepTemp = k
Do While IIf(stepTemp >= 0, i <= toTemp, i >= toTemp)

    j = 0: k = 0 ' Changing j and k has no effect
    Print i;

    i += stepTemp
Loop
Print
```

Dialect Differences

- In the *-lang fb* and *-lang deprecated* dialects, variables declared inside a block are visible only inside the block, and cannot be accessed outside it
- In the *-lang qb* and *-lang fblite* dialects, variables declared inside a block are visible counter if declared, and any temporary variables used to hold a value are visible procedure-wide **scope** as in QB

Differences from QB

- **ByRef** arguments cannot be used as counters.

See also

- **Continue**
- **Do...Loop**
- **Exit**

Format



Formats a number in a specified format

Syntax

```
Declare Function Format ( ByVal numerical_expression As Double,  
formatting_expression As Const String = "" ) As String
```

Usage

```
#include "string.bi"  
result = Format[$]( numerical_expression, formatting_expression
```

Parameters

numerical_expression
number to format
formatting_expression
formatting pattern

Return Value

Format returns a string with the result of the numerical expression formatted by the formatting expression.

The formatting expression is a string that can yield numeric or date-time

Description

To recover meaningful date-time values the numerical expression must be obtained from the appropriate functions.

This function is part of FreeBASIC, however it is not recognized by the `vbcompat.bi` if it is included.

"Numeric Formats"

Symbol	Description
Null string	General format (no formatting)
0	Digit placeholder: If the number has fewer digits than there are zeros (on either side) in the format expression, leading or trailing zeros are displayed. If there are more digits than zeros in the format, the number is rounded. If there are more digits to the left

	the format the digits are all displayed
#	Digit placeholder: Follows the same rules as for the 0 digit except the leading or trailing zeros are not displayed.
.	Placeholder for decimal point. If the format contains only #'s to the left of . then number is not begun with a decimal point.
%	Percentage :The expression is multiplied by 100 and the % character is inserted.
,	Thousands separator. Two adjacent commas, or a comma immediately to the left of a decimal point (whether there is a decimal specified or not) means 'Omit the three digits that fall between the comma and the decimal point, rounding as needed.'
E- E+ e- e+	Scientific format: If a format contains one digit placeholder (0 or #) to the right of a number is displayed in scientific format and an E or e is inserted between the number and the digit placeholder. The number of 0's or #'s to the right determines the number of digits in the exponent. Use a - sign next to negative exponents. Use a E+ or e+ to place a minus sign next to negative exponents.
: ? + \$ () space	Display literal character To display a character other than one of these, precede the character with a backslash (\) or enclose the character(s) in double quotation marks
\	Display next character in format string as it is
text between double quotes	Displays the text inside the double quotes.
:	Time separator is used to separate hours, minutes, and seconds when time values are displayed.
/	The date separator is used to separate day, month, and year when date values are displayed.

"Date-Time formats:"

Symbol	Description
d, dd	Display the day as a one-digit/two-digit number (1-31/01-31)
ddd	Display the day as an abbreviation (Sun-Sat)
dddd	Display the day as a full name (Sunday-Saturday)
dddddd	Display a serial date number as a complete date (including day, month and year)
m, mm	Display the month as a one-digit/two-digit number (1-12/01-12). If immediately followed by a colon, the minute rather than the month is displayed
M, MM	Display the month as a one-digit/two-digit number (1-12/01-12), even if immediately followed by a colon
mmm	Display the month as an abbreviation (Jan-Dec)
mmmm	Display the month as a full name (January-December)
y, yy	Display the year as a two-digit number (00-99)
yyyy	Display the year as a four-digit number (1900-2040)
h, hh	Display the hour as a one-digit/two-digit number (0-23/00-23)
m, mm	Display the minute as a one-digit/two-digit number (0-59/00-59). If not immediately followed by a colon, the month is displayed.

	the month rather than the minute is displayed
n, nn	Display the minute as a one-digit/two-digit number (0-59/00-59), even if no hh
s, ss	Display the second as a one-digit/two-digit number (0-59/00-59)
tttt	Display a time serial number as a complete time, including hour, minute and a
AM/PM (Default), am/pm	Use the 12-hour clock displaying AM or am with any hour before noon, PM between noon and 11:59
A/P, a/p	Use the 12-hour clock displaying A or a with any hour before noon, P or p and 11:59

Example

Sample numeric formats			
Format (fmt)	5	-5	.5
Null String		5	-5
0		5	-5
0.00		5.00	-5.00
#,##0		5	-5
#,##0.00		5.00	-5.00
0%		500%	-500%
0.00%		500.00%	-500.00%
0.00E+00		5.00E+00	-5.00E
0.00E-00		5.00E00	-5.00E
Sample Date And Time Formats			
The following are examples of Date And Time formats:			
Format	Expression	Display	
	m/d/yy	12/7/58	
	d-mmmm-yy	7-December-58	
	d-mmmm	7-December	
	mmmm-yy	December-58	
	h:mm AM/PM	8:50 PM	
	h:mm:ss AM/PM	8:50:35 PM	
	h:mm	20:50	
	h:mm:ss	20:50:35	
	m/d/yy h:mm	12/7/58 20:50	

Dialect Differences

None

Differences from QB

- Does not exist in QB 4.5. This function appeared first in PDS 7

See also

- [\(Print | ?\) Using](#)
- [Str](#)

Returns the decimal part of a number

Syntax

```
Declare Function Frac ( ByVal number As Double ) As Double
Declare Function Frac ( ByVal number As Integer ) As Integer
Declare Function Frac ( ByVal number As UInteger ) As UInteger
Declare Function Frac ( ByVal number As LongInt ) As LongInt
Declare Function Frac ( ByVal number As ULongInt ) As ULongInt
```

Usage

```
result = Frac( number )
```

Parameters

number

the number or expression to get the fraction part of.

Return Value

Returns the fractional part of a number or expression.

Description

Equivalent to: $(number - \text{Fix}(number))$.

For example, `Frac(4.25)` will return `0.25`, and `Frac(-1.75)` will return `-0.75`. For integer types, the value `0` is always returned.

The `Frac` unary **operator** can be overloaded with user defined types.

Example

```
Print frac(10.625)  '' will print 0.625
Print frac(-10.625) '' will print -0.625
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- New to FreeBASIC

See also

- [Fix](#)
- [Operator](#)

Returns the amount of free memory available

Syntax

```
Declare Function Fre ( ByVal value As Long = 0 ) As UInteger
```

Usage

```
result = Fre( [ value ] )
```

Parameters

value

Unused dummy parameter kept for backward compatibility; can be ignored.

Return Value

Returns the amount of free memory, in bytes.

Description

Returns the free memory (ram) available, in bytes.

Example

```
Dim mem As Integer = Fre

Print "Free memory:"
Print
Print mem; " bytes"
Print mem \ 1024; " kilobytes"
Print mem \ (1024 * 1024); " megabytes"
```

Differences from QB

- The "value" argument is not checked, `Free` will always return the free physical memory available

See also

- `Dim`
- `ReDim`
- `Allocate`

Returns a free file number

Syntax

```
Declare Function FreeFile ( ) As Long
```

Usage

```
result = FreeFile
```

Return Value

The next available file number, if any, otherwise zero (0).

Description

Returns the number of the next free file number with valid values 1 to 255. This value is a required argument to **Open** a file. **FreeFile** is useful when the programmer can't keep track of the used file numbers.

Make sure to always close files when no longer needed, otherwise you will not be able to open any files anymore after 255 file numbers are exhausted with **FreeFile**. **FreeFile** will always return the smallest free file number. The file number that **FreeFile** returns is **FreeFile** until that file number is **Open**, or until a smaller file number is **Close**. For the next **FreeFile** call, immediately before its corresponding **Open**, to ensure that the file number is the first.

Example

```
' Create a string and fill it.
Dim buffer As String, f As Integer
buffer = "Hello World within a file."

' Find the first free file number.
f = FreeFile

' Open the file "file.ext" for binary usage, using
Open "file.ext" For Binary As #f
```

```

' Place our string inside the file, using file number
Put #f, , buffer

' Close the file.
Close #f

' End the program. (Check the file "file.ext" upon execution)
End

```

When using multiple `FreeFile` statements, `FreeFile` should be used in

```

Dim fr As Integer, fs As Integer
' The CORRECT way:
fr = FreeFile
Open "file1" For Input As #fr

fs = FreeFile
Open "file2" For Input As #fs

```

As opposed to:

```

Dim fr As Integer, fs As Integer
' The WRONG way:
fr = FreeFile
fs = FreeFile '' fs has taken the same file number as fr

Open "file1" For Input As #fr
Open "file2" For Input As #fs '' error: file number already in use

```

Platform Differences

- On Windows, a file number used in a dynamic link library is not available to the main program. File numbers can not be passed or returned.

executable.

- Besides FreeBASIC's limit of 255 files per program opened at : of opened files, but usually you won't touch it except in DOS, w

Differences from QB

- None

See also

- [Open](#)
- [Put \(File I/O\)](#)
- [Get \(File I/O\)](#)

Function



Defines a procedure returning a value

Syntax

```
[Public|Private] Function identifier [cdecl|pascal|stdcall] [Overload]  
[(parameter_list)] [As return_type] [Static] [Export]  
statements  
...  
{ Return [return_value]}|{Function = return_value}|{identifier  
...  
End Function
```

Parameters

identifier: the name of the function

external_identifier: externally visible (to the linker) name enclosed in an external name

parameter_list: *parameter* [, *parameter* [, ...]]

parameter: [**ByRef**|**ByVal**] *identifier* [**As** *type*] [= *default_value*]

identifier: the name of the variable referenced in the function. If the name is followed by an empty parenthesis.

type: the type of variable

default_value: the value of the argument if none is specified in the call

return_type: the type of variable returned by the function

statements: one or more statements that make up the function body

return_value: the value returned from the function

Description

A function defines a block of code which can be executed with a single call and returns control back to the caller when finished (a return value). There are several reasons to use functions:

- Reduces redundancy in your program
- Enables reuse of code in many programs
- Improves readability of the program
- Improves maintainability of the program
- Makes it easy to extend your program

Access Rights : The **Public** and **Private** keywords specify public or private access rights.

respectively. If neither is given, the function defaults to public access (

Calling Convention : Calling convention, or the order in which arguments are passed in function calls, is specified with the `cdecl`, `pascal` and `stdcall` keyword convention by default (`stdcall`).

Passing Arguments : Functions may receive one or more variables, or listed as *parameters* in the *parameter_list*. The `ByRef` and `ByVal` keywords specify passing by reference or by value, respectively. The argument's type is given by the type in the declaration is given a default value, the parameter is optional. A parameter is optional if it is followed by an identifier with an empty parenthesis. Note that array parameters are not required nor allowed for array parameters. When calling a function with arguments, all arguments must be supplied there too; see the examples.

Overloaded Functions : An overloaded function may share the same name and signature. The `Overload` keyword specifies that a function may be overloaded. The `Overload` keyword must be used prior to any functions that overload them.

Returning values : *return_type* specifies the `data type` returned by a function. If no *return_type* is specified, then the function will return the default data type, which will be Integer, `DefDb1`, `DefStr`, etc. Functions can return values using three methods: returning a value from a function immediately, and returns that value to the caller. Functions can return a value using the `Return` keyword or the function's *identifier* to the desired return value. The `Return` keyword is required to exit, however. `Return` keyword mixed with `Function=` keyword or `Function` keyword is unsupported when returning objects with constructor. `Return` keyword is not supported when returning objects with constructor. `Return` keyword is not supported when returning objects with constructor. Thus, function calls can be made wherever a `Return` statement is used. Parentheses surrounding the argument list are required if there are arguments. Functions can also return references.

Local Variable Preservation : The `Static` keyword specifies that a function's local variables are preserved between function calls. Upon entering a function defined with `Static`, the state of the function was last called.

Example

```
' ' This program demonstrates the declaration of a function and returning a value using Return command
```

```
Declare Function ReturnTen () As Integer

Print ReturnTen () '' ReturnTen returns an integer

Function ReturnTen() As Integer
    Return 10
End Function
```

```
'' This program demonstrates the declaration of a
'' and returning a value using assignment to funct

Declare Function ReturnTen () As Integer

Print ReturnTen () '' ReturnTen returns an integer

Function ReturnTen() As Integer
    ReturnTen = 10
End Function
```

```
'' This program demonstrates function overloading.

'' The overloaded functions must be FIRST.
Declare Function ReturnTen Overload (a As Single)
Declare Function ReturnTen Overload (a As String)
Declare Function ReturnTen (a As Integer) As Integer

Print ReturnTen (10.000!) '' ReturnTen will take a
Print ReturnTen (10)      '' ReturnTen will take a
Print ReturnTen ("10")    '' ReturnTen will take a

Function ReturnTen Overload (a As Single) As Integer
    Return Int(a)
End Function
```

```
Function ReturnTen Overload (a As String) As Integer
    Return Val(a)
End Function
```

```
Function ReturnTen (a As Integer) As Integer
    Return a
End Function
```

'' The following example demonstrates optional parameters

```
Function TestFunc(P As String = "Default") As String
    Return P
End Function
```

```
Print TestFunc("Testing:")
Print TestFunc
```

'' This example shows how to declare and call
'' functions taking array arguments.

```
Function x(b() As Double) As Integer
    x = UBound(b)-LBound(b)+1
End Function
```

```
Dim a(1 To 10) As Double
Print x(a())
Dim c(10 To 20) As Double
Print x(c())
```

Dialect Differences

- In the *-lang fb* dialect, **ByVal** is the default parameter passing c String and user-defined **Types** are passed **ByRef** by default.
- In the *-lang qb* and *-lang fblite* dialects, **ByRef** is the default pa
- In the *-lang qb* dialect, the name of the function must be used Using **Function = ...**" to specify the return value may not be u
- In the *-lang qb* and *-lang fblite* dialects, **Return** may only be u *lang qb*, this must be done explicitly using the **Option Nogosub**

Differences from QB

- Parameters can be optional in FreeBASIC.
- In QBASIC, the return type could only specified with a suffix, n return a built-in type.
- Return value can now be specified by a **Return** statement.
- Function **overloading** is supported in FreeBASIC.
- The return value of functions can be ignored in the calling code

See also

- **Sub**
- **Exit**
- **Return**
- **Declare**
- **Public**
- **Private**

Function (Member)



Declares or defines a member procedure returning a value.

Syntax

```
{ Type | Class | Union } typename
Declare [ Static | Const ] Function fieldname [calling convention specifier] [ Alias external_name ] ( [ parameters ] ) As datatype [
[ Static ]
End { Type | Class | Union }

Function typename.fieldname ( [ parameters ] ) As datatype [
Export ]
statements
End Function
```

Parameters

typename
name of the **Type**, **Class**, or **Union**

fieldname
name of the procedure

external_name
name of field as seen when externally linked

parameters
the parameters to be passed to the procedure

calling convention specifier
can be one of: **cdecl**, **stdcall** or **pascal**

Description

Function members are accessed with **Operator** **.** (**Member Access**) or **Operator** **->** (**Pointer To Member Access**) to call a member procedure that returns a value (a reference can also be returned by specifying **Byref As** *return_type*). The procedure may optionally accept parameters either **ByVal** or **ByRef**. *typename* be overloaded without explicit use of the **Overload** keyword.

typename is the name of the type for which the **Function** method is declared and defined. Name resolution for *typename* follows the same

rules as procedures when used in a **Namespace**.

A hidden **This** parameter having the same type as *typename* is passed to non-static member procedures. **This** is used to access the fields of the **Type**, **Class**, or **Union**.

To access duplicated symbols defined outside the Type, use:
.SomeSymbol (Or ..SomeSymbol if inside a **With..End With** block).

A **Static (Member)** may be declared using the **static** specifier. A **cons (Member)** may be declared using the **const** specifier.

As for a normal **Function**, the return value of a **Function** member can be ignored in the calling code.

Example

```
#include "vbcompat.bi"

Type Date

    value As Double

    Declare Static Function Today() As Date

    Declare Function Year() As Integer
    Declare Function Month() As Integer
    Declare Function Day() As Integer

End Type

Function Date.Today() As Date
    Return Type(Now())
End Function

Function Date.Year() As Integer
    Return ..Year(value)
End Function
```

```
Function Date.Month() As Integer
    Return ..Month(value)
End Function
```

```
Function Date.Day() As Integer
    Return ..Day(value)
End Function
```

```
Dim d As Date = Date.Today
```

```
Print "Year = "; d.Year
Print "Month = "; d.Month
Print "Day = "; d.Day
```

Dialect Differences

- Only available in the *-lang fb* dialect.

See also

- [Class](#)
- [Function](#)
- [Sub \(Member\)](#)
- [Type](#)

Get (Graphics)



Gets a copy of a portion of the current work page or an image buffer

Syntax

```
Get [source,] [STEP](x1, y1) - [STEP](x2, y2), dest
```

Parameters

source

the address of an image buffer.

STEP

indicates that the following co-ordinates are not absolute co-ordinates

[STEP](*x1*, *y1*)

co-ordinates of the upper-left corner of the sub-image to copy. STEP in offsets are relative to the current graphics cursor position.

[STEP](*x2*, *y2*)

co-ordinates of the lower-right corner of the sub-image to copy. STEP in are relative to *x1* and *y1*, respectively.

dest

the address of a previously allocated buffer to store the image data.

Description

get copies a rectangular portion of the current work page specified by *y1*) and (*x2*, *y2*), which represent the upper-left and lower-right corners respectively. STEP specifies that the upper-left co-ordinates are relative graphics pen location, and/or that the lower-right co-ordinates are relative co-ordinates. The new image buffer is formatted to match the current **format**.

dest can be an address, an **array**, or a reference to the first element in receive the new image buffer. This memory must be sufficiently allocated buffer; the number of bytes required varies with the **-lang dialect** user program.

source can be an address, an **array**, or a reference to the first element in an image buffer to retrieve a portion of. *x1*, *y1*, *x2*, *y2*, **Step** and *dest* have in this case.

The co-ordinates of the rectangle are affected by the most recent `winc (Graphics)` statements, and must both be within the current clipping region (`Graphics`), otherwise an illegal function call runtime error will be triggered and will have no effect.

Runtime errors:

`get` throws one of the following **runtime errors**:

(1) *Illegal function call*

- *dest* is an array, but is not big enough to hold the image
- The upper-left or lower-right co-ordinates of the rectangle are outside the current clipping region. See `View (Graphics)`.

Dialect Differences

There are 2 types of buffers (details see `GfxInternalFormats`) depending on the dialect used:

- In the ***-lang fb*** dialect, *dest* receives a new-style image buffer, which consists of a byte image header followed by pixel data which is row-padded to a 16-byte boundary (16 bytes). Use the following formula to calculate the total size required to store the image buffer, where *w* and *h* are the respective width and height of the rectangular portion of the current work page or source image, and *bpp* is the number of bytes per pixel of the current screen mode:

$$\text{size} = 32 + (((w * \text{bpp} + \&hF;) \text{ and not } \&hF;) * h)$$

- In the ***-lang qb*** and ***-lang fblite*** dialects, *dest* receives a QB-style image buffer which consists of a 4-byte image header followed by pixel data which is row-padded to a 16-byte boundary. Use the following formula to calculate the total size required to store the image buffer, where *w* and *h* are the respective width and height of the rectangular portion of the current work page or source image, and *bpp* is the number of bytes per pixel of the current screen mode:

$$\text{size} = 4 + (w * h * \text{bpp})$$

Example

```
#include once "fbgfx.bi"
```

```

'' Setup a 400x300 32bit screen
ScreenRes 400, 300, 32

'' First draw funny stuff...
Line (10,10)-(140,30), RGB(255,255,0), bf
Draw String (30, 20), "Hello there!", RGB(255,0,0)

'' Now capture a 150x50 block from the top-
left of the screen into an image
'' buffer with GET...
Dim As fb.Image Ptr image = ImageCreate(150, 50)
Get (0,0)-(150-1,50-1), image

'' And duplicate it all over the place!
Put (0,50), image
Put (0,100), image
Put (0,150), image
Put (0,200), image
Put (0,250), image
Put (150,0), image
Put (150,50), image
Put (150,100), image
Put (150,150), image
Put (150,200), image
Put (150,250), image

'' And a frame around a whole screen..
Line (0,0)-(400-1,300-1), RGB(255,255,0), b

'' Now get the whole screen...
Dim As fb.Image Ptr big = ImageCreate(400, 300)
Get (0,0)-(400-1,300-1), big

'' And display that "screenshot" as if it was scre
Dim As Integer x = -350
While ((Inkey() = "") And (x < 350))
    ScreenLock
        Cls

```

```
        Put (x,0), big
ScreenUnlock

Sleep 100, 1

x += 10
Wend
```

See also

- **Put (Graphics)**
- **Get (File I/O)**
- **Screen (Graphics)**
- **Window**
- **View (Graphics)**
- **Internal graphics formats**

Get (File I/O)



Reads data from a file to a buffer

Syntax

```
Get #filenum As Long, [position As LongInt], ByRef data As Any [  
Get #filenum As Long, [position As LongInt], data As String [, ,  
Get #filenum As Long, [position As LongInt], data() As Any [, ,
```

Usage

```
Get #filenum, position, data [, [amount] [, bytesread ] ]  
varres = Get (#filenum, position, data [, [amount] [, bytesread
```

Parameters

filenum

The value passed to **open** when the file was opened.

position

The position where the read must start. If the file was opened **For Random** reading starts at the present file pointer position. The position is 1-based.

If *position* is omitted or zero (0), file reading will start from the current

data

The buffer where data is written. It can be a numeric variable, a string operation will try to fill completely the variable, unless the **EOF** is reached. When getting arrays, *data* should be followed by an empty pair of brackets. **EOF** is not allowed.

When getting **strings**, the number of bytes read is the same as the number of bytes in the buffer.

Note: If you want to read values into a buffer, you should NOT pass a pointer to the buffer. (This can be done by dereferencing the pointer with **operator** to overwrite the pointer variable, not the memory it points to.)

amount

Makes **Get** read *amount* consecutive variables from file to memory, i.e. memory starting at *data*'s memory location. If *amount* is omitted it defaults to the number of bytes in the buffer.

An unsigned integer variable to accept the result of the number of bytes read.

Return Value

Zero (0) on success; non-zero on error. Note: if **EOF** (end of file) is reached, the return value is non-zero.

actually read can be checked by passing a *bytesread* variable.

Description

Reads binary data from a file to a buffer variable

`get` can be used as a function, and will return 0 on success or an error

For files opened in **Random** mode, the size in bytes of the data to read is

Example

```
Dim Shared f As Integer

Sub get_integer()

    Dim buffer As Integer ' Integer variable

    ' Read an Integer (4 bytes) from the file into
    Get #f, , buffer

    ' print out result
    Print buffer
    Print

End Sub

Sub get_array()

    Dim an_array(0 To 10-1) As Integer ' array of

    ' Read 10 Integers (10 * 4 = 40 bytes) from th
    Get #f, , an_array()

    ' print out result
    For i As Integer = 0 To 10-1
        Print an_array(i)
    Next
```

```

    Print

End Sub

Sub get_mem

    Dim pmem As Integer Ptr

    ' allocate memory for 5 Integers
    pmem = Allocate(5 * SizeOf(Integer))

    ' Read 5 integers (5 * 4 = 20 bytes) from the
    Get #f, , *pmem, 5 ' Note pmem must be derefer

    ' print out result using [] Pointer Indexing
    For i As Integer = 0 To 5-1
        Print pmem[i]
    Next
    Print

    ' free pointer memory to prevent memory leak
    Deallocate pmem

End Sub

' Find the first free file file number.
f = FreeFile

' Open the file "file.ext" for binary usage, using
Open "file.ext" For Binary As #f

    get_integer()

    get_array()

    get_mem()

' Close the file.
Close #f

```

```

' Load a small text file to a string

Function LoadFile(ByRef filename As String) As String

    Dim h As Integer
    Dim txt As String

    h = FreeFile

    If Open( filename For Binary Access Read As #h) Then

        If LOF(h) > 0 Then

            txt = String(LOF(h), 0)
            If Get( #h, ,txt ) <> 0 Then txt = ""

        End If

        Close #h

        Return txt

    End Function

Dim ExampleStr As String
ExampleStr = LoadFile("smallfile.txt")
Print ExampleStr

```

Differences from QB

- `get` in FB can read full arrays as in VB or, alternatively, read a 1
- `get` can be used as a function in FB, to find the success/error c
- FB allows the *bytesread* parameter, to check how many bytes l

See also

- **Get (Graphics)** different usage of same keyword
- **Put (File I/O)**
- **Open**
- **Close**
- **Binary**
- **Random**
- **FreeFile**
- **File I/O methods comparison**

GetJoystick



Reads buttons and axis information from attached gaming devices

Syntax

```
Declare Function GetJoystick ( ByVal id As Long, ByRef buttons As Single = 0, ByRef a2 As Single = 0, ByRef a3 As Single = 0, ByRef a4 As Single = 0, ByRef a5 As Single = 0, ByRef a6 As Single = 0, ByRef a7 As Single = 0, ByRef a8 As Single = 0 ) As Boolean
```

Usage

```
result = GetJoystick( id[, buttons[, a1[, a2[, a3[, a4[, a5[, a6
```

Parameters

id
the device id number (0 - 15)

buttons
the button status

a1
first axis value

a2
second axis value

a3
third axis value

a4
fourth axis value

a5
fifth axis value

a6
sixth axis value

a7
seventh axis value

a8
eighth axis value

Return Value

0 on success or 1 on failure. All of the axis positions are returned in float

Description

GetJoystick will retrieve the button state, and the axis positions for *u* by *id*, a number between 0 and 15. Buttons are stored in a similar matrix *buttons* representing a button.

A single precision value between -1.0 and 1.0 is returned for each value controller, a value of -1000.00 is returned.

GetJoystick will return 0 upon successful completion; It will return 1 upon specifying an illegal joystick number, specifying a joystick which does

Example

```
Screen 12

Dim x As Single
Dim y As Single
Dim buttons As Integer
Dim result As Integer
Dim a As Integer

Const JoystickID = 0

'This line checks to see if the joystick is ok.

If GetJoystick(JoystickID,buttons,x,y) Then
    Print "Joystick doesn't exist or joystick error"
    Print
    Print "Press any key to continue."
    Sleep
    End
End If

Do
    result = GetJoystick(JoystickID,buttons,x,y)

    Locate 1,1
```

```
Print ;"result:";result;" x:" ;x;" y:";y;" But
'This tests to see which buttons from 1 to 27
For a = 0 To 26
    If (buttons And (1 Shl a)) Then
        Print "Button ";a;" pressed.    "
    Else
        Print "Button ";a;" not pressed."
    End If
Next a
Loop
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- Screen (Graphics)
- SetMouse
- GetMouse
- MultiKey

Returns the ascii code of the first key in the keyboard buffer

Syntax

```
Declare Function GetKey ( ) As Long
```

Usage

```
result = GetKey
```

Return Value

The value of the ascii code returned.

Description

It returns the ascii code of the first key in the keyboard buffer. The key removed from the buffer. If no key is present, `GetKey` waits for it. For extended keys (returning two characters), the extended code is returned the first byte, and the regular code is returned in the second byte. (see example below)

The key read is not echoed to the screen.

For a keyword not stopping the program if no key is at the buffer see `GetKey` or `MultiKey`.

Example

```
Dim As Integer foo
Do
    foo = GetKey
    Print "total return: " & foo

    If( foo > 255 ) Then
        Print "extended code: " & (foo And &hff)
        Print "regular code: " & (foo Shr 8)
```

```
Else
    Print "regular code: " & (foo)
End If
Print
Loop Until foo = 27
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__Getkey`.

Differences from QB

- New to FreeBASIC

See also

- `GetMouse`
- `Inkey`
- `Input()`
- `MultiKey`

GetMouse



Retrieves the status of the mouse pointing device

Syntax

```
Declare Function GetMouse ( ByRef x As Integer, ByRef y As Integer, ByRef wheel As Integer = 0 ) As Long
```

Usage

```
result = GetMouse (x, y [, [ wheel ] [, [ buttons ] [, [ clip ]]
```

Parameters

x
x coordinate value
y
y coordinate value
wheel
scroll wheel value
buttons
button status
clip
clip status

Return Value

0 on success, or 1 on error (for example because the mouse is outside the window)

Description

GetMouse retrieves the mouse position and buttons status; information not available, all variables will contain the -1 value.

If in console mode, the *x* and *y* coordinates are the character cell coordinates 0, 0. If the mouse moves out of the console window, **GetMouse** returns -1. In console mode and fullscreen, the scroll wheel value is not returned.

If in graphics mode, *x* and *y* will always be returned in pixel coordinate case; custom coordinates system set via **View** or **Window** do not affect the result.

If the mouse runs off the graphic window, all values are set to -1 and 1 for the buttons and wheel if the return value of the function is not also

Wheel is the mouse wheel counter; rotating the wheel away from you r decrease. At program startup or when a new graphics mode is set via mouse wheels for a given platform, in which case 0 is always returned

Buttons stores the buttons status as a bitmask: bit 0 is set if left mouse middle mouse button / wheel is down.

Clip stores the mouse clipping status; if 1, the mouse is currently clipped

Example

```
Dim As Integer x, y, buttons, res
' Set video mode and enter loop
ScreenRes 640, 480, 8
Do
    ' Get mouse x, y and buttons. Discard wheel position
    res = GetMouse (x, y, , buttons)
    Locate 1, 1
    If res <> 0 Then '' Failure

#ifdef __FB_DOS__
    Print "Mouse or mouse driver not available"
#else
    Print "Mouse not available or not on window"
#endif

    Else
        Print Using "Mouse position: ###:###  Buttons: "
        If buttons And 1 Then Print "L";
        If buttons And 2 Then Print "R";
        If buttons And 4 Then Print "M";
        Print " "
    End If
Loop While Inkey = ""
End
```

```

'Example 2: type-union-type structure
Type mouse
  As Integer res
  As Integer x, y, wheel, clip
  Union
    buttons As Integer
  Type
    Left:1 As Integer
    Right:1 As Integer
    middle:1 As Integer
  End Type
End Union
End Type

Screen 11
Dim As mouse m

Do
  m.res = GetMouse( m.x, m.y, m.wheel, m.buttons
ScreenLock
Cls
Print Using "res = #"; m.res
Print Using "x = ###; y = ###; wheel = +###; c
Print Using "buttons = ##; left = #; middle =
ScreenUnlock
Sleep 10, 1
Loop While Inkey = ""

```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the `Integer`.

Platform Differences

- On Win32, scroll wheel changes are not guaranteed to be detected
- In DOS, the "clip" value has no relevance. Additionally the wheel is not supported by the mouse driver. See also FaqDOS.

Differences from QB

- New to FreeBASIC

See also

- **ScreenRes** setting graph mode by resolution
- **Screen (Graphics)** setting mode the QB-like way
- **SetMouse**
- **MultiKey**
- **GetJoystick**

Control flow statement to use a section of code and return.

Syntax

```
GoSub label
```

Description

Execution jumps to a subroutine marked by a line label. Always use **Return** to exit a **GoSub**, execution will continue on next statement after **GoSub**.

The line label where **GoSub** jumps must be in the same main/function/sub block as **GoSub**. All the variables in the subroutine are shared with the block, no arguments can be used. For this reason **GoSub** is considered bad programming practice as it can generate unreadable and untraceable code. It is better to use **Sub** or **Function** instead.

Example

```
' ' Compile with -lang qb
'$lang: "qb"

GoSub message
End

message:
Print "Welcome!"
Return
```

Dialect Differences

- Only available in the *-lang qb* and *-lang fblite* dialects.

- `GoSub` support is disabled by default in the *-lang fblite* unless the `Option Gosub` statement is used.

Differences from QB

- None when using the *-lang qb* dialect.

See also

- `Goto`
- `Return`
- `Sub`
- `Function`
- `Option Gosub`

Goto



Control flow statement to jump to another part of the program

Syntax

```
Goto label
```

Description

Jumps code execution to a line label.

For better source code readability, overuse of `Goto` should be avoided in favor of more modern structures such as `Do...Loop`, `For...Next`, `Sub` and `Function`.

Example

```
Goto there

backagain:
  End

there:
  Print "Welcome!"
  Goto backagain
```

```
' ' Compile with -lang qb or fblite

'$lang: "qb"

1 Goto 3
2 End
3 Print "Welcome!"
4 Goto 2
```

Dialect Differences

- Line numbers are allowed only in the *-lang qb* and *-lang deprecated* dialects.

Differences from QB

- None

See also

- [GoSub](#)
- [Sub](#)
- [Function](#)

Returns the hexadecimal of the given number

Syntax

```
Declare Function Hex ( ByVal number As UByte ) As String
Declare Function Hex ( ByVal number As UShort ) As String
Declare Function Hex ( ByVal number As ULong ) As String
Declare Function Hex ( ByVal number As ULongInt ) As String
Declare Function Hex ( ByVal number As Const Any Ptr ) As String

Declare Function Hex ( ByVal number As UByte, ByVal digits As
Long ) As String
Declare Function Hex ( ByVal number As UShort, ByVal digits As
Long ) As String
Declare Function Hex ( ByVal number As ULong, ByVal digits As
Long ) As String
Declare Function Hex ( ByVal number As ULongInt, ByVal digits As
Long ) As String
Declare Function Hex ( ByVal number As Const Any Ptr, ByVal
digits As Long ) As String
```

Usage

```
result = Hex[$]( number [, digits ] )
```

Parameters

number

A number or expression evaluating to a number. A floating-point number will be converted to a **LongInt**.

digits

Optional number of digits to return.

Return Value

A **String** containing the unsigned hexadecimal representation of *number*.

Description

Returns the unsigned hexadecimal string representation of the integer

number. Hexadecimal digits range from 0-9, or A-F.

If you specify *digits* > 0, the result string will be exactly that length. It will be truncated or padded with zeros on the left, if necessary.

The length of the string will not go longer than the maximum number of digits required for the type of *number* (8 for a **Long**, 16 for a **LongInt**).

If you want to do the opposite, i.e. convert a hexadecimal string back into a number, the easiest way to do it is to prepend the string with "&H;", and convert it to an integer type, using a function like **CInt**, similarly to a normal numeric string. E.g. **CInt("&HFF;")**

Example

```
'54321 is D431 in hex  
Print Hex(54321)  
Print Hex(54321, 2)  
Print Hex(54321, 5)
```

will produce the output:

```
D431  
31  
0D431
```

Dialect Differences

- The string type suffix "\$" is obligatory in the **-lang qb** dialect.
- The string type suffix "\$" is optional in the **-lang fblite** and **-lang fb** dialects.

Differences from QB

- In QBASIC, there was no way to specify the number of digits

returned.

- The size of the string returned was limited to 32 bits, or 8 hexadecimal digits.

See also

- **Bin**
- **Oct**
- **ValInt**
- **ValLng**

Gets the second byte of the operand.

Syntax

```
#define HiByte( expr ) ((Cast(UInteger, expr) And &h0000FF00;) S
```

Usage

```
result = HiByte( expr )
```

Parameters

expr

A numeric expression, converted to an **UInteger** value.

Return Value

Returns the value of the high byte of the low 16bit word of *expr*.

Description

This macro converts the numeric expression *expr* to an **UInteger** value representing the value of its second byte - that is the most-significant significant (low) 16bit word of *expr*.

Example

```
Dim N As UInteger

'Note there are 16 bits
N = &b1010101110000001
Print "N is
Print "The binary representation of N is
Print "The most significant byte (MSB) of N is
Print "The least significant byte (LSB) of N is
Print "The binary representation of the MSB is
Print "The binary representation of the LSB is
```

Sleep

The output would look like:

```
N Is 43905
The Binary representation of N Is 1010101110000001
The most significant Byte (MSB) of N Is 171
The least significant Byte (LSB) of N Is 129
The Binary representation of the MSB Is 10101011
The Binary representation of the LSB Is 10000001
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- [LoByte](#)
- [LoWord](#)
- [HiWord](#)

Gets the second 16bit word of the operand.

Syntax

```
#define HiWord( expr ) ((Cast(UInteger, expr) and &hFFFF0000;) S
```

Usage

```
result = HiWord( expr )
```

Parameters

expr

A numeric expression, converted to an **UInteger** value.

Return Value

Returns the value of the high 16bit word of the low 32bit dword of *expr*

Description

This macro converts the numeric expression *expr* to an **UInteger** value representing the value of its second 16bit word - that is the most-significant (high) 16bit word of the low 32bit dword of *expr*.

Example

```
Dim N As UInteger

'Note there are 32 bits
N = &b10000000000000000111111111111111

Print "N is"
Print "The binary representation of N is"
Print "The most significant word (MSW) of N is"
Print "The least significant word (LSW) of N is"
Print "The binary representation of the MSW is"
```

```
Print "The binary representation of the LSW is  
Sleep
```

The output would look like:

```
N Is 2147614719  
The Binary representation of N Is 10000000000000001111  
The most significant word (MSW) of N Is 32769  
The least significant word (LSW) of N Is 65535  
The Binary representation of the MSW Is 10000000000000001  
The Binary representation of the LSW Is 1111111111111111
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- [LoByte](#)
- [HiByte](#)
- [LoWord](#)

Hour



Gets the hour of day from a **Date Serial**

Syntax

```
Declare Function Hour ( ByVal date_serial As Double ) As Long
```

Usage

```
#include "vbcompat.bi"  
result = Hour( dateserial )
```

Parameters

date_serial
the date serial

Return Value

Returns the hour from a variable containing a date in **Date Serial** form

Description

The compiler will not recognize this function unless `vbcompat.bi` is inc

Example

```
#include "vbcompat.bi"  
  
Dim ds As Double = DateSerial(2005, 11, 28) + Time  
Print Format(ds, "yyyy/mm/dd hh:mm:ss "); Hour(ds)
```

Differences from QB

- Did not exist in QB. This function appeared in PDS and VBDO:

See also

- **Date Serials**

Control flow statement for conditional branching

Syntax

```
If expression Then [statement(s)] [Else [statement(s)]] [End If]  
or  
If expression Then : [statement(s)] [Else [statement(s)]] : End  
or  
If expression Then  
[statement(s)]  
[ ElseIf expression Then ]  
[statement(s)]  
[ Else ]  
[statement(s)]  
End If
```

Description

If...Then is a way to make decisions. It is a mechanism to execute code or provide alternative code to execute based on more conditions.

expression can be one of several forms:

- a conditional expression, for example:
`x = 5`
- multiple conditions separated by logical bit-wise operators
`x >= 5 And x <= 10`
- multiple conditions separated by logical short-circuit operators
`y <> 0 AndAlso x \ y = 1`
(in this case, "`x \ y = 1`" will only be evaluated if "`y <> 0`" is True)
- any numerical expression, in which case a value of zero or a non-zero value represents True

Both multi-line and single-line **ifs** can be nested. In the latter case, the control where nested **ifs** begin and end.

In the *-lang fb* and *-lang fblite* dialects, colons (:) can be used instead of blocks on a single line.

Example

```
' ' Here is a simple "guess the number" game using  
  
Dim As Integer num, guess  
  
Randomize  
num = Int(Rnd * 10) + 1 'Create a random number between 1 and 10  
  
Print "guess the number between 1 and 10"  
  
Do 'Start a loop  
  
    Input "Guess"; guess 'Input a number from the user  
  
    If guess > 10 OrElse guess < 1 Then 'The user's guess is out of range  
        Print "The number can't be greater than 10 or less than 1"  
    ElseIf guess > num Then 'The user's guess is too high  
        Print "Too high"  
    ElseIf guess < num Then 'The user's guess is too low  
        Print "Too low"  
    ElseIf guess = num Then 'The user guessed the number  
        Print "Correct!"  
        Exit Do 'Exit the loop  
    End If  
  
Loop 'Go back to the start of the loop
```

Dialect Differences

- In the *-lang qb* and *-lang fblite* dialects, variables declared inside a block have a wide **scope** as in QB
- In the *-lang fb* and *-lang deprecated* dialects, variables declared inside a block are only accessible inside the block, and cannot be accessed outside it.
- In the *-lang qb* dialect, if there is a new line or a single-line comment, the line will be multi-line. A colon, a **Rem** or any other statement will result in a multi-line.

- In the *-lang fb* and *-lang fblite* dialects, if there is a new line, a **Rem** statement directly after THEN, then the IF will be multi-line single-line IF.

Differences from QB

- END IF was not supported in single-line IFs in QBASIC.

See also

- **Do...Loop**
- **#if**
- **Select Case**

Conditional function that returns one of two values.

Syntax

```
IIf ( condition, expr_if_true, expr_if_false )
```

Parameters

condition

The condition to test.

A non-zero value evaluates as true, while a value of zero evaluates as

expr_if_true

An expression to evaluate and return if *condition* is true.

It must return:

- a numeric value, which can be an integer, floating point, or pointer,
- or a string value,
- or an UDT value.

expr_if_false

An expression to evaluate and return if *condition* is false.

It must be same type as *expr_if_true* (either numeric, either string or

Description

IIf returns a different numeric or string or UDT value depending of the conditional expression. Its typical use is in the middle of an expression to put a conditional in the middle.

IIf only evaluates the expression that it needs to return. This saves time and can be useful to prevent evaluating expressions that might be invalid depending on the *condition*.

Warning: The ability to accept mixed numeric types, strings and UDTs was added from the fbc version 0.90.

Example

```
Dim As Integer a, b, x, y, z
a = (x + y + IIf(b > 0, 4, 7)) \ z
```

is equivalent to:

```
Dim As Integer a, b, x, y, z, temp
If b > 0 Then temp = 4 Else temp = 7
a = (x + y + temp) \ z
```

```
Dim As Integer I
I = -10
Print I, IIf(I>0, "positive", IIf(I=0, "null", "ne
I = 0
Print I, IIf(I>0, "positive", IIf(I=0, "null", "ne
I = 10
Print I, IIf(I>0, "positive", IIf(I=0, "null", "ne
Sleep
```

```
Type UDT1
  Dim As Integer I1
End Type

Type UDT2 Extends UDT1
  Dim As Integer I2
End Type

Dim As UDT1 u1, u10 = (1)
Dim As UDT2 u2, u20 = (2, 3)

u1 = IIf(0, u10, u20)
Print u1.I1
u1 = IIf(1, u10, u20)
Print u1.I1

u2 = IIf(0, u10, u20)
```

```
Print u2.I1; u2.I2
'u2 = Iif(1, u10, u20) ''Invalid assignment/conver
Sleep
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- [If...Then](#)

ImageConvertRow



Converts a row of image data into another color depth

Syntax

```
Declare Sub ImageConvertRow ( ByVal src As Any Ptr, ByVal src_bpp  
ByVal dst_bpp As Long, ByVal width As Long, ByVal isrgb As Long
```

Usage

```
ImageConvertRow( src, src_bpp, dst, dst_bpp, width [, isrgb ] )
```

Parameters

src

The address of the start of the source row. The source can either be a 32 bits per pixel, or a paletted image with a bit depth of 1-8 bits per pixel; only work properly if you are in a screen mode that is using the correct conversion.

src_bpp

The number of bits per pixel in the source row. 1-8, 24 and 32.

dst

The address of the start of the destination row. The image can be a full image or a palette. If the source is a paletted image, the destination can also be a palette.

dst_bpp

The number of bits per pixel in the destination row. Valid values for this are 1-8, 24 and 32.

width

The length of the row in pixels.

isrgb

A value of zero indicates that the Red and Blue channels are the other way around. A value of one indicates that the Red and Blue channels are swapped in this switch if you want the Red and Blue channels to be swapped in the destination image.

Description

Copies the row of an image from one memory location to another, converting the color depth of each pixel to match the destination image.

Example

```
#include "fbgfx.bi"
```

```

#if __FB_LANG__ = "fb"
Using FB
#endif

Const As Integer w = 64, h = 64
Dim As IMAGE Ptr img8, img32
Dim As Integer x, y

'' create a 32-bit image, size w*h:
ScreenRes 1, 1, 32, , GFX_NULL
img32 = ImageCreate(w, h)

If img32 = 0 Then Print "Imagecreate failed on img32"

'' create an 8-bit image, size w*h:
ScreenRes 1, 1, 8, , GFX_NULL
img8 = ImageCreate(w, h)

If img8 = 0 Then Print "Imagecreate failed on img8"

'' fill 8-bit image with a pattern
For y = 0 To h - 1
    For x = 0 To w - 1
        PSet img8, (x, y), 56 + (x + y) Mod 24
    Next x
Next y

'' open a graphics window in 8-bit mode, and PUT t
ScreenRes 320, 200, 8
WindowTitle "8-bit color mode"
Put (10, 10), img8

Sleep

```

```

'' copy the image data into a 32-bit image
Dim As Byte Ptr p8, p32
Dim As Integer pitch8, pitch32

#ifdef ImageInfo '' older versions of FB don't ha
#define GETPITCH(img_) IIf(img_>Type=PUT_HEADER_M
>old.width*img_>old.bpp)
#define GETP(img_) CPtr(Byte Ptr,img_)+IIf(img_-
>Type=PUT_HEADER_NEW,SizeOf(PUT_HEADER),SizeOf(_OL
pitch8 = GETPITCH(img8): p8 = GETP(img8)
pitch32 = GETPITCH(img32): p32 = GETP(img32)
#else
ImageInfo( img8, , , , pitch8, p8 )
ImageInfo( img32, , , , pitch32, p32 )
#endif

For y = 0 To h - 1
    ImageConvertRow(@p8 [ y * pitch8 ], 8, _
                    @p32[ y * pitch32], 32, _
                    w)
Next y

'' open a graphics window in 32-bit mode and PUT t
ScreenRes 320, 200, 32
WindowTitle "32-bit color mode"
Put (10, 10), img32

Sleep

'' free the images from memory:
ImageDestroy img8
ImageDestroy img32

```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- **ScreenRes**
- **Get (Graphics)**
- **Put (Graphics)**
- **ImageCreate**
- **ImageDestroy**
- **ImageInfo**

ImageCreate



Allocates and initializes storage for an image

Syntax

```
Declare Function ImageCreate ( ByVal width As Long, ByVal height  
Ulong = transparent_color ) As Any Ptr  
Declare Function ImageCreate ( ByVal width As Long, ByVal height  
Ulong = transparent_color, ByVal depth As Long ) As Any Ptr
```

Usage

```
result = ImageCreate( width, height [, [ color ][, depth ]] )
```

Parameters

width

The desired width, in number of pixels.

height

The desired height, in number of pixels.

color

The pixel value to fill the area of the image.

depth

The desired color depth, in bits per pixel.

Return Value

If the image could not be created, NULL (0) is returned, otherwise, the returned. **ImageCreate** must be called after graphic mode initialization,

Consequently, in case of **shared** variable declaration, **ImageCreate** can initializer, even inside an **udt** (in member field or constructor), because shared variable) is set at the start of the program before any user code allocation call must be in a separated executable instruction, and after initialization.

Description

Both procedures attempt to allocate memory for an image of the specified size. If successful, **ImageCreate** returns a pointer to the image. Otherwise, NULL (0) is returned. Otherwise, an image of that size is created.

filling the entire area of pixels with the value *color*. If not specified, *color* is the transparent color for the current graphics screen, which can be found by calling `ScreenControl`. In any case, the address of the image is returned, which the user must be destroyed using `ImageDestroy`.

The first procedure creates an image with a color depth matching that of the screen, which can be found by calling `ScreenControl`. The second procedure creates an image with a color depth of *depth*, in bits per pixel. For both procedures, the image is used in drawing procedures while in any screen mode -- and across modes. The color depth of the image matches that of the graphics screen.

`ImageCreate` is the recommended way to allocate memory for new images, structures, etc. -- while documented, it may change from version to version and the calculation of the sizes involved is error-prone. However, `ImageInfo` can be used for other things, the size, in bytes, of an existing image, allowing memory to be copied, or to be read from or written to a file or device.

`Get (Graphics)` can be used to initialize an image using pre-allocated memory.

Example

```
' ' Create a graphics screen.
ScreenRes 320, 200, 32

' ' Create a 64x64 pixel image with a darkish green
Dim image As Any Ptr = ImageCreate( 64, 64, RGB(0, 0, 128))

If image = 0 Then
    Print "Failed to create image."
    Sleep
End If

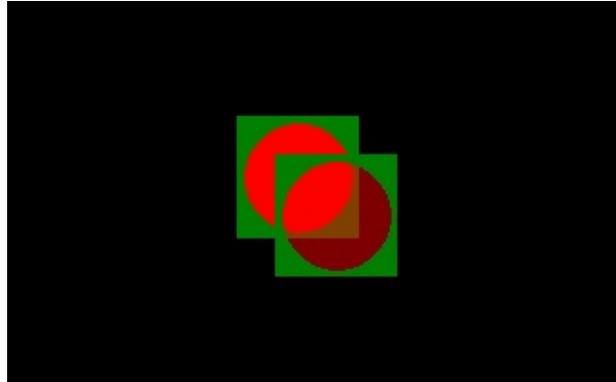
' ' Draw a semi-transparent, red circle in the center
Circle image, (32, 32), 28, RGBA(255, 0, 0, 128),, 1

' ' Draw the image onto the screen using various blitting
Put (120, 60), image, PSet
```

```
Put (140, 80), image, Alpha
```

```
'' Destroy the image.  
ImageDestroy image
```

```
Sleep
```



Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- `ImageDestroy`
- `ImageInfo`
- `Get (Graphics)`
- `Internal pixel formats`

ImageDestroy



Destroys and deallocates storage for an image

Syntax

```
Declare Sub ImageDestroy ( ByVal image As Any Ptr )
```

Usage

```
ImageDestroy( image )
```

Parameters

image

The address of the image to destroy.

Description

Destroys the image pointed to by *image*, which must be an address returned from a call to [ImageCreate](#).

Calling `ImageDestroy` on a null pointer induces no action.

Example

See [ImageCreate](#) for an example on using `ImageDestroy`.

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__Imagedestroy`.

Differences from QB

- New to FreeBASIC

See also

- [ImageCreate](#)

ImageInfo



Retrieves information about an image

Syntax

```
Declare Function ImageInfo ( ByVal image As Any Ptr, ByRef width  
As Integer = 0, ByRef pitch As Integer = 0, ByRef pixdata As Any
```

Usage

```
result = ImageInfo( image [, [width] [, [height] [, [bypp] [, [p
```

Parameters

image

The address of the image.

width

Stores the width of the image, in pixels.

height

Stores the height of the image, in pixels.

bypp

Stores the bytes per pixel of the image - i.e. the size of a single pixel,

pitch

Stores the pitch of the image - i.e. the size of each scanline (row), in k because the scanlines may be padded to allow them to be aligned be

pixdata

Stores the address of the start of the first scanline of the image.

size

Stores the size of the image in memory, in bytes.

Return Value

If *image* doesn't point to a valid image, one (1) is returned. Otherwise, appropriate values, and zero (0) is returned.

Description

ImageInfo provides various information about an image, such as its di information you need to directly access all the pixel data in the pixel b

It can also provide the size of the image in memory, which is useful for writing an image to a file.

Example

```
' ' pixelptr(): use imageinfo() to find the pointer
' ' returns null on error or x,y out of bounds
Function pixelptr(ByVal img As Any Ptr, ByVal x As Integer, ByVal y As Integer) As Any Ptr

    Dim As Integer w, h, bypp, pitch
    Dim As Any Ptr pixdata
    Dim As Integer success

    success = (ImageInfo(img, w, h, bypp, pitch, pixdata))

    If success Then
        If x < 0 Or x >= w Then Return 0
        If y < 0 Or y >= h Then Return 0
        Return pixdata + y * pitch + x * bypp
    Else
        Return 0
    End If

End Function

' ' usage example:

' ' 320*200 graphics screen, 8 bits per pixel
ScreenRes 320, 200, 8

Dim As Any Ptr ip ' ' image pointer

Dim As Byte Ptr pp ' ' pixel pointer (use byte for pointer)

ip = ImageCreate(32, 32) ' ' create an image (32*32)

If ip <> 0 Then
```

```

'' draw a pattern on the image
For y As Integer = 0 To 31

    For x As Integer = y - 5 To y + 5 Step 5

        '' find the pointer to pixel at x,y po
        '' note: this is inefficient to do for
        pp = pixelptr(ip, x, y)

        '' if success, plot a value at the pix
        If (pp <> 0) Then *pp = 15

    Next x

Next y

'' put the image and draw a border around it
Put (10, 10), ip, PSet
Line (9, 9)-Step(33, 33), 4, b

'' destroy the image to reclaim memory
ImageDestroy ip

Else
    Print "Error creating image!"
End If

Sleep

```

```

'' Create 32-bit graphics screen and image.
ScreenRes 320, 200, 32
Dim image As Any Ptr = ImageCreate( 64, 64 )

Dim pitch As Integer
Dim pixels As Any Ptr

```

```

'' Get enough information to iterate through the p
If 0 <> ImageInfo( image, ,,, pitch, pixels ) Then
    Print "unable to retrieve image information."
    Sleep
    End
End If

'' Draw a pattern on the image by directly manipul
For y As Integer = 0 To 63
    Dim row As ulong Ptr = pixels + y * pitch

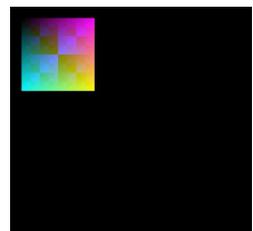
    For x As Integer = 0 To 63
        row[x] = RGB(x * 4, y * 4, (x Xor y) * 4)
    Next x
Next y

'' Draw the image onto the screen.
Put (10, 10), image

ImageDestroy( image )

Sleep

```



Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- **ImageCreate**
- **ImageDestroy**
- **ImageConvertRow**
- **Get (Graphics)**
- **Put (Graphics)**
- **Internal pixel formats**

Operator Imp (Implication)



Returns the bitwise-and (implication) of two numeric values

Syntax

```
Declare Operator Imp ( ByRef lhs As T1, ByRef rhs As T2 ) As Ret
```

Usage

```
result = lhs Imp rhs
```

Parameters

lhs

The left-hand side expression.

T1

Any numeric or boolean type.

rhs

The right-hand side expression.

T2

Any numeric or boolean type.

Ret

A numeric or boolean type (varies with *T1* and *T2*).

Return Value

Returns the bitwise-implication of the two operands.

Description

This operator returns the bitwise-implication of its operands, a logical operation that results in a value with bits set depending on the bits of the operands (for conversion of a boolean to an integer, false or true boolean value becomes 0 or -1 integer value).

The truth table below demonstrates all combinations of a boolean-implication operation:

Lhs Bit	Rhs Bit	Result

0	0	1
1	0	0
0	1	1
1	1	1

No short-circuiting is performed - both expressions are always evaluated.

The return type depends on the types of values passed. **Byte**, **UByte** and floating-point type values are first converted to **Integer**. If the left and right-hand side types differ only in signedness, then the return type is the same as the left-hand side type (τ_1), otherwise, the larger of the two types is returned. Only if the left and right-hand side types are both **Boolean**, the return type is also **Boolean**.

This operator can be overloaded for user-defined types.

Example

```
Dim As UByte a, b, c
a = &b00001111
b = &b01010101
c = a Imp b ' c = &b11110101
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- None

See also

- **Operator Truth Tables**

Specifies an interface to be implemented by a user-defined type
Note: Stub page. Even though this keyword is reserved already, interfaces are not implemented yet.

Syntax

```
Type typename Implements interface  
...  
End Type
```

Description

Example

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__Implements`.

Differences from QB

- New to FreeBASIC

See also

- [Type](#)
- [Extends](#)

Import



External linkage attribute for public data located in DLL's

Syntax

```
Extern Import symbolname[( subscripts)] [ Alias "aliasname"] [ A  
[, ...]
```

Description

Is used only on Win32 platforms with the **Extern** keyword and is needed for global variables in DLLs. This is due to the level of indirection on any pointer dereference.

Example

```
/* mydll.c :  
   compile with  
   gcc -Shared -Wl,--strip-all -o mydll.dll myc  
*/  
__declspec( dllexport ) Int MyDll_Data = 0x1234;
```

```
/' import.bas :  
  Compile with  
  fbc import.bas  
  '/  
  #inlib "mydll"  
  
  Extern Import MyDll_Data Alias "MyDll_Data" As Int  
  
  Print "&h" + Hex( MyDll_Data )  
  
  ' Output:  
  ' &h1234
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the `__Import`.

Differences from QB

- New to FreeBASIC

See also

- [Extern](#)

Returns a string representing the first key waiting in the keyboard buffer

Syntax

Declare Function Inkey () As String

Usage

```
result = Inkey[$]
```

Return Value

The first character found in the keyboard buffer, or an empty string ("")

Description

Peeks into the keyboard buffer and returns a **String** representation of removed from the buffer, and is not echoed to the screen. If the keybc returned without waiting for keys.

If the key is in the ASCII character set, a one-character **String** consist "extended" one (numeric pad, cursors, functions) a two-character **Str**: character (*See dialect differences below*)

The Shift, Ctrl, Alt, and AltGr keys can't be read independently by this (although, perhaps obviously, Shift-A will be reported by **Inkey** differer the above).

See also **Input()** or **GetKey**, or **Sleep** to wait for a key press if the keyb

Example

```
Print "press q to quit"  
Do  
    Sleep 1, 1  
Loop Until Inkey = "q"
```

```

#if __FB_LANG__ = "qb"
#define EXTCHAR Chr$(0)
#else
#define EXTCHAR Chr(255)
#endif

Dim k As String

Print "Press a key, or Escape to end"
Do

    k = Inkey$

    Select Case k

        Case "A" To "Z", "a" To "z": Print "Letter"
        Case "1" To "9":           Print "Number"

        Case Chr$(32): Print "Space"

        Case Chr$(27): Print "Escape"

        Case Chr$(9): Print "Tab"

        Case Chr$(8): Print "Backspace"

        Case Chr$(32) To Chr$(127)
            Print "Printable character: " & k

        Case EXTCHAR & "G": Print "Up Left / Home"
        Case EXTCHAR & "H": Print "Up"
        Case EXTCHAR & "I": Print "Up Right / PgUp"

        Case EXTCHAR & "K": Print "Left"
        Case EXTCHAR & "L": Print "Center"
        Case EXTCHAR & "M": Print "Right"
    End Select
Loop

```

```

Case EXTCHAR & "O": Print "Down Left / End
Case EXTCHAR & "P": Print "Down"
Case EXTCHAR & "Q": Print "Down Right / Pg

Case EXTCHAR & "R": Print "Insert"
Case EXTCHAR & "S": Print "Delete"

Case EXTCHAR & "k": Print "Close window /

Case EXTCHAR & Chr$(59) To EXTCHAR & Chr$(
    Print "Function key: F" & Asc(k, 2) -

Case EXTCHAR & Chr$(133) To EXTCHAR & Chr$(
    Print "Function key: F" & Asc(k, 2) -

Case Else
    If Len(k) = 2 Then
        Print Using "Extended character: c
    ElseIf Len(k) = 1 Then
        Print Using "Character chr$(###)";
    End If

End Select

If k = Chr$(27) Then Exit Do

Sleep 1, 1

Loop

```

Dialect Differences

- The extended character is `chr(255)` in the *-lang fb* and *-lang f*
 - In the *-lang qb* dialect, the extended character depends used, the extended character is `chr(0)`. If it is reference
 - In all other dialects, the extended char is always `chr(255)`

- The string type suffix \$ is optional in the *-lang fblite* and *-lang*

Differences from QB

- None in the *-lang qb* dialect.
- QBasic returned a `chr(0)` as the first character for an extended character in the *-lang fb* and *-lang fblite* dialects.

See also

- `Sleep`
- `GetKey`
- `Input()`
- `MultiKey`

Returns a value at a hardware port.

Syntax

```
Declare Function Inp ( ByVal port As UShort ) As Integer
```

Usage

```
value = Inp(port)
```

Parameters

port
Port number to read.

Return Value

The value at the specified port.

Description

This function retrieves the value at 'port' and returns immediately.

Example

```
' ' Turn off PC speaker  
Out &h61, Inp(&h61) And &hfc
```

Platform Differences

- In the Windows and Linux versions three port numbers (&H3C7;, &H3C8;, &H3C9;) are hooked by the graphics library when a graphics mode is in use to emulate QB's VGA palette handling. This use is deprecated; use **Palette** to retrieve and set palette colors.

- Using true port access in the Windows version requires the program to install a device driver for the present session. For that reason, Windows executables using hardware port access should be run with administrator permits each time the computer is restarted. Further runs don't require admin rights as they just use the already installed driver. The driver is only 3K in size and is embedded in the executable.

See also

- [Out](#)
- [Wait](#)
- [Palette](#)

Reads a list of values from the keyboard

Syntax

```
Input [;] ["prompt" ,|; ] variable_list
```

Parameters

prompt

an optional string literal that is written to the screen as a prompt. If it is a semicolon (;), a question mark ("? ") will be appended to the prompt. If it is a comma, nothing will be appended.

variable_list

a list of comma-separated variables used to hold the values read from the keyboard

Description

Reads a list of values from the keyboard up until the first carriage return. The values are converted from their string representation into the corresponding type. Characters are echoed to the screen as they are typed.

If there is more than one value in the input list, then the input line will be split up by delimiters - commas (,) after strings, or commas and whitespace after numbers. Surrounding whitespace will be trimmed from string values. If an input value contains a space, it must be wrapped in quotes ("...") to prevent it being split up. For inputting to a single string without delimiting, **Line Input** should be used.

The *prompt* - if any - is written to the screen at the current cursor location. The characters read are echoed to the screen immediately following the prompt. If no characters are echoed at the current cursor location.

The optional leading semicolon (;) after **Input** is similar to the optional **Print** statement: the cursor will remain on the same line after all of the values have been echoed, otherwise, the cursor will move to the beginning of the next line.

If more values are read than are listed in the variable list, extra values are discarded. If fewer values are read (i.e. the user presses enter before inputting all values), the remaining variables in the list are left unchanged.

variables will be initialized - numeric variables to zero (0), and string variables to an empty string ("").

Numeric values are converted using methods similar to the procedure using the most appropriate function for the number format, converting characters as possible.

Input has a limited edit capacity: it allows to erase characters using the better user interface is needed, a custom input routine should be used.

Example

Example #1

```
Dim n As String, a As Integer
Input "Enter [Name, Age]: ", n, a
Print n
Print a
```

Example #2

```
Dim As Double a, b
Dim As String yn

Do

    Input "Please enter a number: ", a
    Input ; "And another: ", b
    Print , "Thank you"
    Sleep 500
    Print
    Print "The total is "; a + b
    Print

Do
    Input "Would you like to enter some more numbers? (y/n): ", yn
    yn = LCase(yn)
```

```
Loop Until yn = "y" Or yn = "n"  
Loop While LCase(yn) = "y"
```

Differences from QB

- If the user inputs the wrong number of values, or if it expects a number and gets a string that is not a valid number, then QBASIC issues the message "Invalid number from start", and does not continue further until it receives a valid number.
- QB does not treat space as a delimiter when inputting a number.

See also

- **Input #**
- **Input()**
- **Line Input**

Input (File Mode)



Specifies text file to be opened for input mode

Syntax

```
Open filename for Input [Encoding encoding_type] [Lock lock_type]  
[#] filenum
```

Parameters

filename

file name to open for input

encoding_type

indicates encoding type for the file

lock_type

locking to be used while the file is open

filenum

unused file number to associate with the open file

Description

A file mode used with **open** to open a text file for reading.

This mode allows to read sequentially lines of text with **Line Input #**, read comma separated values with **Input #**.

Text files can't be simultaneously read and written in FreeBASIC, so if functions are required on the same file, it must be opened twice.

filename must be a string expression resulting in a legal file name in the OS, without wildcards. The file will be sought for in the present directory unless the *filename* contains a path. If the file does not exist, an error is issued. The pointer is set at the first character of the file.

Encoding_type indicates the Unicode **Encoding** of the file, so characters are correctly read. If omitted, "ascii" encoding is defaulted. Only little endian character encodings are supported at the moment.

- "utf8",
- "utf16"

- "utf32"
- "ascii" (the default)

Lock_type indicates the way the file is locked for other processes, it is

- **Read** - the file can be opened simultaneously by other processes but not for writing
- **Write** - the file can be opened simultaneously by other processes, but not for writing
- **Read Write** - the file cannot be opened simultaneously by other processes (the default)

filenum is a valid FreeBASIC file number (in the range 1..255) not being used for any other file presently open. The file number identifies the file for subsequent file operations. A free file number can be found using the [FreeFile](#) function.

Example

```

Dim ff As UByte
Dim randomvar As Integer
Dim name_str As String
Dim age_ubyte As UByte

ff = FreeFile
Input "What is your name? ", name_str
Input "What is your age? ", age_ubyte
Open "testfile" For Output As #ff
Write #ff, Int(Rnd(0)*42), name_str, age_ubyte
Close #ff
randomvar=0
name_str=""
age_ubyte=0

Open "testfile" For Input As #ff
Input #ff, randomvar, name_str, age_ubyte
Close #ff

Print "Random Number was: ", randomvar

```

```
Print "Your name is: " + name_str  
Print "Your age is: " + Str(age_ubyte)
```

```
'File outputted by this sample will look like this  
'minus the comment of course:  
'23, "Your Name", 19
```

Differences from QB

See also

- **Append**
- **Open**
- **Output**

Input



Reads a list of values from a text file

Syntax

```
Input # filenum, variable_list
```

Parameters

filenum

a file number of a file or device opened for **Input**

variable_list

a list of variables used to hold the values read

Description

Reads from a text file through a bound file number a delimiter-separated set of values and writes them in reading order into the variables in *variable_list*. If a variable is numeric the read value is converted from its string representation into the corresponding type.

Numeric values are converted in a similar way to the procedures **val** and **valLng**, using the most appropriate function for the number format.

Delimiters may be commas or line breaks. Whitespace is also treated as a separator after numbers. A string including a comma or a whitespace must be surrounded by double quotes.

To read an entire line into a string, use **Line Input** instead.

Write # can be used to create a file readable with **Input #**.

Example

```
Dim a As Integer
Dim b As String
Dim c As Single

Open "myfile.txt" For Output As #1
```

```
Write #1, 1, "Hello, World", 34.5
Close #1

Open "myfile.txt" For Input As #1
Input #1, a, b, c
Close #1
Print a, b, c
```

Differences from QB

- QB has a bug in INPUT # that causes it to read past the end of the line if it does not find a matching end-quote when reading a string. If you are porting QB code that relies upon this bug, you may need to edit your data files to remove newlines from inside quoted strings, or to use a custom function to piece back together the multiline string.

See also

- **Input**
- **Line Input #**
- **Write #**
- **Open**
- **Input (File Mode)**

Input()



Reads a number of characters from console or file

Syntax

```
Declare Function Input ( n As Integer ) As String  
Declare Function Input ( n As Integer, filename As Integer ) As  
String
```

Usage

```
result = Input[$]( n [, [#]filename ] )
```

Parameters

n
Number of bytes to read.

filename
File number of a bound file or device.

Return Value

Returns a **string** of the characters read.

Description

Reads a number of characters from the console, or a bound file/device specified by *filename*.

The first version waits for and reads *n* characters from the keyboard buffer. Extended keys are not read. The characters are not echoed to the screen.

The second version waits for and reads *n* characters from a file or device. The file position is updated.

Example

```
Print "Select a color by number"
```

```
Print "1. blue"  
Print "2. red"  
Print "3. green"  
Dim choice As String  
Do  
    choice = Input(1)  
Loop Until choice >= "1" And choice <= "3"
```

Differences from QB

- None

See also

- Winput()
- GetKey
- Inkey

Locates the first occurrence of a substring or character within a string

Syntax

```
Declare Function InStr ( ByRef str As Const String, [ Any ] ByRe
Declare Function InStr ( ByRef str As Const WString, [ Any ] ByR
Declare Function InStr ( ByVal start As Integer, ByRef str As Co
Declare Function InStr ( ByVal start As Integer, ByRef str As Co
```

Usage

```
first = InStr( [ start, ] str, [ Any ] substring )
```

Parameters

str

The string to be searched.

substring

The substring to find.

start

The position in *str* at which the search will begin. The first character s

Return Value

The position of the first occurrence of *substring* in *str*.

Description

Locates the position of the first occurrence of a substring or character search begins at the first character.

Zero (0) is returned if: either *substring* is not found, either *str* or *subs*

If the **Any** keyword is specified, **InStr** returns the first occurrence of an

Example

```
' It will return 4
```

```
Print InStr("abcdefg", "de")
```

' It will return 0

```
Print InStr("abcdefg", "h")
```

' It will search for any of the characters "f", "b"

```
Print InStr("abcdefg", Any "fbc")
```

```
Dim test As String
```

```
Dim idx As Integer
```

```
test = "abababab"
```

```
idx = InStr(test, "b")
```

```
Do While idx > 0 'if not found loop will be skipped
```

```
Print ""b"" at " & idx
```

```
idx = InStr(idx + 1, Test, "b")
```

```
Loop
```

'A Unicode example:

```
dim text as wstring*20
```

```
text = "Привет, мир!"
```

```
print instr(text,"ет") ' displays 5
```

Platform Differences

- The wide-character string version of `InStr` is not supported for

Differences from QB

- QB returns *start* if *search* is a zero length string.
- QB does not support Unicode.

See also

- InStrRev
- Mid (Function)

Locates the last occurrence of a substring or character within a string

Syntax

```
Declare Function InStrRev ( ByRef str As Const String, [ Any ] B  
substring As Const String, ByVal start As Integer = -1 ) As Inte  
Declare Function InStrRev ( ByRef str As Const WString, [ Any ]  
substring As Const WString, ByVal start As Integer = -1 ) As Int
```

Usage

```
last = InStrRev( str, [ Any ] substring [, start ] )
```

Parameters

str

The string to be searched.

substring

The substring to find.

start

The position in *str* at which the search will begin. The first character is position 1.

Return Value

The position of the last occurrence of *substring* in *str*.

Description

Locates the position of the last occurrence of a substring or character string. If *start* parameter is not given or is -1, the search begins at the character.

Zero (0) is returned if: either *substring* is not found, or either *str* or *substring* are empty strings, or *start* is less than 1 (except for -1), or *start* is greater than the length of *str*.

If the **Any** keyword is specified, **InStrRev** returns the last occurrence of character in *substring*.

Example

```
' It will return 4  
Print InStrRev("abcdefg", "de")
```

```
' It will return 0  
Print InStrRev("abcdefg", "h")
```

```
Dim test As String  
Dim idx As Integer  
  
test = "abababab"  
idx = InStrRev(test, "b")  
  
Do While idx > 0 'if not found loop will be skipped  
    Print ""b"" at " & idx  
    idx = InStrRev(test, "b", idx - 1)  
Loop
```

```
'A Unicode example:  
dim text as wstring*20  
text = "Привет, мир!"  
print instrrev(text,"ет") ' displays 5
```

Platform Differences

- The wide-character string version of `InStrRev` is not supported on the Windows target.

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the `qb` keyword.

`__Instrrev.`

Differences from QB

- New to FreeBASIC

See also

- `InStr`
- `Mid (Function)`

Returns the floor of a number

Syntax

```
Declare Function Int ( ByVal number As Single ) As Single
Declare Function Int ( ByVal number As Double ) As Double
Declare Function Int ( ByVal number As Integer ) As Integer
Declare Function Int ( ByVal number As UInteger ) As UInteger
Declare Function Int ( ByVal number As LongInt ) As LongInt
Declare Function Int ( ByVal number As ULongInt ) As ULongInt
```

Usage

```
result = Int( number )
```

Parameters

number
the floating-point number to round

Return Value

Returns the floor of *number*, i.e. the largest integer that is less than or equal to it.

Description

`Int` returns the floor of *number*. For example, `Int(4.9)` will return `4.0`, and `Int(-1.3)` will return `-2.0`. For integer types, the number is returned unchanged.

The `Int` unary **operator** can be overloaded with user defined types.

Example

```
Print Int(1.9)  '' will print 1
Print Int(-1.9) '' will print -2
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- None

See also

- [Fix](#)
- [CInt](#)
- [Operator](#)

Integer



Standard data type: 32-bit or 64-bit signed, same size as `SizeOf(Any Ptr)`

Syntax

```
Dim variable As Integer
Dim variable As Integer<bits>
```

Parameters

bits

A numeric constant expression indicating the size in bits of integer desired. The values allowed are 8, 16, 32 or 64.

Description

32-bit or 64-bit signed whole-number data type, depending on the platform.

Integer is the main data type FreeBASIC uses for integer math and bitwise operations. It is the default type for number literals.

If an explicit bit size is given, a data type is provided that can hold values from $-1LL \text{ sh}1 \text{ (bits-1)}$ up to $(1LL \text{ sh}1 \text{ (bits-1)}) - 1$.

Example

```
#if __FB_64BIT__
  Dim x As Integer = &H8000000000000000
  Dim y As Integer = &H7FFFFFFFFFFFFFFF
  Print "Integer Range = "; x; " to "; y
#else
  Dim x As Integer = &H80000000
  Dim y As Integer = &H7FFFFFFF
  Print "Integer Range = "; x; " to "; y
#endif
```

Dialect Differences

- In the *-lang fb* and *-lang fblite* dialects, the `Integer` data type is 32-bit.
- In the *-lang qb* dialect, the `Integer` data type is 16-bit, regardless of platform.

Differences from QB

- The ability to select a bit size is new to FreeBASIC
- The `INTEGER` type is always 16 bits wide in QB.

See also

- `Long`
- `LongInt`
- `UInteger`
- `CInt`

Clause in the **select case** statement block.

Syntax

Case Is *expression*

Description

Is specifies that a particular case inside a Select Case block will be evaluated based on an expression including the greater than (>) or less than (<) operator and a value.

See also

- **Select Case**
- **Operator Is**

Operator Is (Run-Time Type Information)



Checks whether an object is compatible to a type derived from its compile-time type.

Syntax

result = *expression* **Is** *typename*

Parameters

expression

The expression to check, an object of a type that is directly or indirectly derived from the type of *expression*.

typename
The child type to check for. This type must be directly or indirectly derived from the type of *expression* (the compile-time type of the object).

Return Value

Returns negative one (-1) if the expression is an object of real-type *ty* or a base-type derived from the *expression* type, or zero (0) if it's an object of a type derived from the *typename* type.

Description

The **Is** operator is a binary operator that checks whether an object is of a type derived from the type of *expression* at run-time. Because **Is** relies on run-time type information, it can only be used for types that are derived from the built-in **Object** type. The compiler disallows uses of **Is** for types that can be solved at compile-time.

The **Is** operator is successful not only for the real-type (the "lowest"), but also for types derived from it, as long as they are still below the type of *expression* (the compile-time type). To determine the real-type, all possibilities from lowest to highest must be checked.

Extending the built-in **Object** type allows to add an extra hidden vtable of the **Type**. The vtable is used to access information for run-time type checking by the **Is** operator.

Example

```

Type Vehicle extends object
  As String Name
End Type

Type Car extends Vehicle
End Type

Type Cabriolet extends Car
End Type

Type Bike extends Vehicle
End Type

Sub identify(ByVal p As object Ptr)
  Print "Identifying:"

  '' Not a Vehicle object?
  If Not (*p Is Vehicle) Then
    Print , "unknown object"
    Return
  End If

  '' The cast is safe, because we know it's a Ve
  Print , "name: " & CPtr(Vehicle Ptr, p)->Name

  If *p Is Car Then
    Print , "It's a car"
  End If

  If *p Is Cabriolet Then
    Print , "It's a cabriolet"
  End If

  If *p Is Bike Then
    Print , "It's a bike"
  End If
End Sub

```

```
Dim As Car ford
ford.name = "Ford"
identify(@ford)

Dim As Cabriolet porsche
porsche.name = "Porsche"
identify(@porsche)

Dim As Bike mountainbike
mountainbike.name = "Mountain Bike"
identify(@mountainbike)

Dim As Vehicle v
v.name = "some unknown vehicle"
identify(@v)

Dim As Object o
identify(@o)
```

Differences from QB

- New to FreeBASIC

See also

- **Extends**
- **Object**
- **Is (Select Case)**

Tests if a string can be converted to a **Date Serial**

Syntax

```
Declare Function IsDate ( ByRef stringdate As Const String ) As Long
```

Usage

```
#include "vbcompat.bi"  
result = IsDate( stringdate )
```

Parameters

stringdate
the string to test

Return Value

Returns non-zero (-1) if the date string can be converted to a **Date Serial**, otherwise returns zero (0).

Description

Date strings must be in the format set in the regional settings of the O to be considered valid dates.

IsDate(**Date**) will return non-zero (-1) only if the regional settings specify the same date format that QB used.

The compiler will not recognize this function unless `vbcompat.bi` or `datetime.bi` is included.

Example

```
#include "vbcompat.bi"  
  
Dim s As String, d As Integer
```

```

Do
  Print
  Print "Enter a date: "

  Line Input s

  If s = "" Then Exit Do

  If IsDate( s ) = 0 Then
    Print ""; s; " is not a valid date"
  Else
    d = DateValue( s )
    Print "year = "; Year( d )
    Print "month = "; Month( d )
    Print "day = "; Day( d )
  End If

Loop

```

Differences from QB

- New to FreeBASIC

See also

- **Date Serials**
- **DateSerial**
- **TimeValue**
- **DateValue**

Isredirected



Checks whether stdin or stdout is redirected to a file

Syntax

```
Declare Function IsRedirected ( ByVal is_input As Long = 0 ) As
```

Usage

```
#include "fbio.bi"  
result = IsRedirected( is_input )
```

Parameters

is_input
A **Long** indicating the type of information to return.

Return Value

Returns non-zero (-1) if stdin or stdout is redirected, otherwise returns

Description

IsRedirected checks whether stdin or stdout is redirected to a file, inc

If *is_input* is equal to non-zero (-1), **IsRedirected** checks stdin.

If *is_input* is equal to zero (0), **IsRedirected** checks stdout.

Example

```
' ' A Windows based example, just for the use princ  
' ' Self-  
sufficient example, using his own .exe file as dum  
  
#include "fbio.bi"  
  
' ' Quotation marks wrapping for compatibility with  
Dim As String pathExe = "" & ExePath & ""  
Dim As String fileExe = Mid(Command(0), InStrRev(C
```

```

Dim As String redirection = " < "" & Command(0)
If LCase(Right(Command(0), 4)) = ".exe" Then
    redirection &= ""
Else
    redirection &= ".exe""
End If

If Command() = "" Then ' First process without s
    ' Check stdin redirection
    Print "First process without stdin redirection:
    ' Creation of asynchronous second process with
    Shell("start /d " & pathExe & " /b " & fileExe &
    ' Waiting for termination of asynchronous secur
    Sleep
ElseIf Command() = "secondprocess" Then ' Second
    ' Check stdin redirection
    Print "Second process with stdin redirection :
End If

```

Differences from QB

- New to FreeBASIC.

See also

- [Reset\(Streamno\)](#)

Deletes a file from disk / storage media.

Syntax

```
Declare Function Kill ( ByRef filename As Const String ) As Long
```

Usage

```
result = Kill( filename )
```

Parameters

filename

The *filename* is the name of the disk file to delete. If the file is not in the current directory, the *path* must also be given as *path/file*.

Return Value

Returns zero (0) on success, or non-zero on error.

Description

`kill` deletes a file from disk / storage media.

Example

```
Dim filename As String = "file.ext"  
Dim result As Integer = Kill( filename )  
  
If result <> 0 Then Print "error trying to kill "
```

Platform Differences

On some platforms, `kill` may be able to remove folders and read-only files. The `fail` here is not currently defined. It may be necessary to check the `fail` before deleting, and decide accordingly whether you want to try `kill`ing it.

Differences from QB

- KILL can optionally be used as function in FreeBASIC.

See also

- [Shell](#)
- [RmDir](#)

Returns the lower bound of an array's dimension

Syntax

```
Declare Function LBound ( array() As Any, ByVal dimension As Int  
= 1 ) As Integer
```

Usage

```
result = LBound( array [, dimension ] )
```

Parameters

array

an array of any type

dimension

the dimension to get lower bound of

Return Value

Returns the lower bound of an array's dimension.

Description

LBound returns the lowest value that can be used as an index into a particular dimension of an array.

Array dimensions are numbered from one (1) to *n*, where *n* is the total number of dimensions. If *dimension* is not specified, **LBound** will return lower bound of the first dimension.

If *dimension* is zero (0), **LBound** returns 1, corresponding to the lower bound of the array dimensions 1..*n*. **UBound** returns *n*, the number of dimensions in this case. This can be used to detect the array's number of dimensions.

For any other (non-zero) *dimension* values outside of the valid range 1..*n*, **LBound** returns 0. **UBound** returns -1 in this case. This can be used to detect whether a certain dimension exists in the array, and also works when

on an empty array which does not have any valid dimensions.

Thus, for empty dynamic arrays, we get:

- `Lbound(array) = 0` and `Ubound(array) = -1` (dimension does not exist)
- `Lbound(array, 0) = 1` and `Ubound(array, 0) = 0` (zero dimensions)
- `@array(Lbound(array)) = 0` (no data buffer allocated)

Example

```
Dim array(-10 To 10, 5 To 15, 1 To 2) As Integer
Print LBound(array) 'returns -10
Print LBound(array, 2) 'returns 5
Print LBound(array, 3) 'returns 1
```

See also

- [UBound](#)
- [Static](#)
- [Dim](#)
- [ReDim](#)

LCASE



Returns a lower case copy of a string

Syntax

```
Declare Function LCASE ( ByRef str As Const String, ByVal mode As Long = 0 ) As String
Declare Function LCASE ( ByRef str As Const WString, ByVal mode As Long = 0 ) As WString
```

Usage

```
result = LCASE[$]( str [ , mode ] )
```

Parameters

str

String to convert to lowercase.

mode

The conversion mode: 0 = current locale, 1 = ASCII only

Return Value

Lowercase copy of *str*.

Description

Returns a copy of *str* with all of the letters converted to lower case.

If *str* is empty, the null string ("") is returned.

Example

```
Print LCASE("AbCdEfG")
```

Output:

```
abcdefg
```

Platform Differences

- The wide-character string version of `Lcase` is not supported for DOS target.

Dialect Differences

- The string type suffix "\$" is obligatory in the *-lang qb* dialect.
- The string type suffix "\$" is optional in the *-lang fblite* and *-lang fb* dialects.

Differences from QB

- QB does not support Unicode.

See also

- `UCase`

Returns the leftmost substring of a string

Syntax

```
Declare Function Left ( ByRef str As Const String, ByVal n As Integer ) As String  
Declare Function Left ( ByRef str As Const WString, ByVal n As Integer ) As WString
```

Usage

```
result = Left[$]( str, n )
```

Parameters

str

The source string.

n

The number of characters to return from the source string.

Return Value

Returns the leftmost substring from *str*.

Description

Returns the leftmost *n* characters starting from the left (beginning) of *str*. If *str* is empty, then the null string ("") is returned. If $n \leq 0$ then the null string ("") is returned. If $n > \text{len}(str)$ then the entire source string is returned.

Example

```
Dim text As String = "hello world"  
Print Left(text, 5)
```

will produce the output:

```
hello
```

An Unicode example:

```
dim text as wstring*20  
text = "Привет, мир!"  
print left(text, 6) 'displays "Привет"
```

Platform Differences

- DOS does not support the wide-character string version of `Left`.

Dialect Differences

- The string type suffix "\$" is obligatory in the *-lang qb* dialect.
- The string type suffix "\$" is optional in the *-lang fblite* and *-lang fb* dialects.

Differences from QB

- QB does not support Unicode.

See also

- [Right](#)
- [Mid \(Function\)](#)

Returns the length of an expression or data type

Syntax

```
Declare Function Len ( ByRef expression As String ) As Integer
```

```
Declare Function Len ( ByRef expression As ZString ) As Integer
```

```
Declare Function Len ( ByRef expression As WString ) As Integer
```

```
Declare Operator Len ( ByRef expression As datatype ) As datatype
```

```
Declare Function Len ( datatype ) As Integer
```

Usage

```
result = Len( expression )
```

or

```
result = Len( DataType )
```

Parameters

expression

An expression of any type.

datatype

A **DataType**.

Return Value

Returns the size of an expression or **DataType** in bytes.

Description

Len returns the length of an expression or the size of a **DataType**, in bytes.

In the first form, if *expression* is of type **String**, **wString** or **zString**, the length of the string in characters will be returned. If the expression is of a user defined type, an **operator Len** compatible with that data type is called. Otherwise, the size of the *expression*'s data type in bytes is returned.

In the second form, if *expression* is `ZString` or `wString`, the size in bytes of an ASCII or Unicode character is returned, respectively. If *datatype* is `String`, the size of the string descriptor type is returned.

If there is both a user defined type and a variable visible with the same name in the current scope, the user defined type takes precedence over the variable. To ensure that the `Len` takes the variable instead of the user defined type, wrap the argument to `Len` with parentheses to force it to be seen as an expression. For example `Len((variable))`.

The `Len` unary **operator** can be overloaded with user defined types.

Example

```
Print Len("hello world") 'returns "11"
Print Len(Integer) ' returns 4

Type xyz
  a As Integer
  b As Integer
End Type

Print Len(xyz) ' returns 8
```

Dialect Differences

- `Len` only allows expressions in the ***-lang qb*** dialect.
- Can be used with built-in types and user-defined types in the ***-lang fb*** and ***-lang fblite*** dialects.

Differences from QB

- Can be used with built-in types and user-defined types in the ***-lang fb*** and ***-lang fblite*** dialects.
- None in the ***-lang qb*** dialect.

See also

- [SizeOf](#)

Indicates the assignment operator.

Syntax

```
Let variable = value
or
Let( variable1 [, variable2 [, ... ]] ) = udt
or
Operator typename.Let ( [ ByRef | ByVal ] rhs As datatype )
statements
end operator
```

Description

Command intended to help the programmer to distinguish an assignment statement (e.g. `Let a = 1`) from an equality test (e.g. `If a = 1 then ...`). As the compiler does not require it, it is usually omitted

Let can be used as a left-hand side operator to assign the members of a user defined type to multiple variables. See [Operator Let\(\) \(Assignment\)](#)

Let is used with operator overloading to refer the assignment operator. See [Operator Let \(Assignment\)](#)

Example

```
' ' Compile with -lang fblite or qb
#lang "fblite"

' these two lines have the same effect:
Let x = 100
x = 100
```

Dialect Differences

- The use of **Let** to indicate an assignment statement (*Let variable = expr*) is not allowed in the **-lang fb** dialect.
- The UDT to multi-variable Let assignment is only available in the **-lang fb** dialect.
- Overloading of operators is not available in the **-lang qb** and **-lang fblite** dialects.

Differences from QB

- None in the **-lang fb** dialect.
- The Let operator is new to FreeBASIC.
- The UDT to multi-variable Let assignment is new to FreeBASIC

See also

- **Operator [=]>** (Assignment)
- **Operator Let** (Assignment)
- **Operator Let()** (Assignment)
- **Operator**

Specifies the library where a sub or function can be found as part of a declaration.

Syntax

```
Declare { Sub | Function } proc_name Lib "libname" [ Alias "symbol" ] ( arguments list ) As return_type
```

```
Extern "mangling" lib "libname"  
declarative statements  
end Extern
```

```
Type T  
As Integer dummy  
Declare Constructor Lib "libname" [ Alias "symbol_name" ] ( argument list )  
end Type
```

Description

In **Sub** or **Function** declarations, and also in class method declarations (including constructors and destructors), **Lib** indicates the library containing the function. Libraries specified in this way are linked in as if **#Inclib** "**Libname**" or **-l libname** had been used.

Lib can also be used with **Extern ... End Extern Blocks** to specify all declarations inside.

Example

```
' ' mydll.bas  
' ' compile with:  
' ' fbc -dll mydll.bas  
  
Public Function GetValue() As Integer Export  
    Function = &h1234  
End Function
```

```
Declare Function GetValue Lib "mydll" () As Integer
```

```
Print "GetValue = &h"; Hex(GetValue())
```

```
' Expected Output :
```

```
' GetValue = &h1234
```

Differences from QB

- New to FreeBASIC

See also

- **Declare**
- **#inlib**

Line (Graphics)



Draws a line

Syntax

```
Line [target,] [[STEP] (x1, y1)]-[STEP] (x2, y2) [, [color][, [B  
or  
Line - (x2, y2) [, [color][, [B|BF][, style]]]
```

Parameters

target

specifies buffer to draw on

STEP

indicates that the starting coordinates are relative

(*x1*, *y1*)

starting coordinates of the line

STEP

indicates that ending coordinates are relative

(*x2*, *y2*)

ending coordinates of the line

color

the color attribute.

B|BF

specifies box or box filled mode

style

line style

Description

Graphics statement that draws a straight line or a box between two points on the current work page set via [ScreenSet](#), or onto the buffer [Get/Put](#) buffer.

Line coordinates are affected by custom coordinates system set via [Window](#) statements, and respect clipping rectangle set by [View \(Graphics\)](#). If by the STEP keyword, the coordinates are assumed to be relative to the top-left corner. If the B flag is specified, a rectangle will be drawn instead of a line, with coordinates of the opposite rectangle corners. If BF is specified, a filled rectangle will be drawn.

color denotes the color attribute, which is mode specific (see [color](#) at

omitted, the current foreground color as set by the `color` statement is

`style`, if specified, allows styled line drawing; its value is interpreted as a bit mask to use it to skip pixel drawing. Starting at $(x1, y1)$, the most significant bit of the pixel is drawn, if 0, it's skipped. This repeats for all the line pixels until the bit being reused when the 16 bits are all checked.

When `Line` is used as `Line - (x2, y2)`, a line is drawn from the current coordinates specified by `Line`. Alternatively, `Point` can be used to get the

Example

```
' ' draws a diagonal red line with a white box, and  
ScreenRes 13  
Line (20, 20)-(300, 180), 4  
Line (140, 80)-(180, 120), 15, b  
Line - ( 200, 200 ), 15  
Sleep 3000
```

```
' Draws 2 lines with 2 different line styles in 2  
ScreenRes 320, 240  
  
Line (10, 100)-  
(309, 140), 4, B, &b1010101010101010 ' red box with  
  
Line (20, 115)-(299, 115), 9, , &b11110000111111  
Line (20, 125)-(299, 125), 10, , &b00000000111100  
  
Sleep
```

Differences from QB

- `target` is new to FreeBASIC

See also

- **Circle**
- **Window**
- **View (Graphics)**

Line Input



Reads one line of input from the keyboard

Syntax

```
Line Input [;] [promptstring {;|,} ] stringvariable
```

Parameters

promptstring
prompt to display before waiting for input
stringvariable
variable to receive the line of text

Description

Reads a line of text from the keyboard and stores it in a string variable

Example

```
Dim x As String  
Line Input "Enter a line:", x  
Print "You entered '"; x; "'"
```

Differences from QB

- QBASIC only allowed literal strings for the prompt text. FreeBASIC allows any variable or constant string expression.

See also

- `Line Input #`
- `Input`

Line Input



Reads one line of text from a file

Syntax

`Line Input #file number, string_variable`

Parameters

file number

file number of an file opened for **Input**

string_variable

variable to receive the line of text

Description

Reads a line from an open text file (opened for **Input** through a bound file number) and stores it in a string variable.

A line of text ends at, but does not include the end of line characters. An end of line character may be the LF character (**Chr**(10)) or the CRLF character pair (**Chr**(13,10)).

Example

```
Dim s As String

Open "myfile.txt" For Output As #1
Print #1, "Hello, World"
Close #1

Open "myfile.txt" For Input As #1
Line Input #1, s
Close #1
Print s
```

Differences from QB

- None

See also

- **Input #**
- **Open**
- **Input (File Mode)**

Gets the lowest byte of the operand.

Syntax

```
#define LoByte( expr ) (Cast(UInteger, expr) And &h000000FF;)
```

Usage

```
result = LoByte( expr )
```

Parameters

expr

A numeric expression, converted to an **UInteger** value.

Return Value

Returns the value of the low byte of *expr*.

Description

This macro converts the numeric expression *expr* to an **UInteger** value representing the value of its least-significant (low) byte.

Example

```
Dim N As UInteger

'Note there are 16 bits
N = &b1010101110000001
Print "N is
Print "The binary representation of N is
Print "The most significant byte (MSB) of N is
Print "The least significant byte (LSB) of N is
Print "The binary representation of the MSB is
Print "The binary representation of the LSB is
Sleep
```

The output would look like:

```
N Is 43905
The Binary representation of N Is 1010101110000001
The most significant Byte (MSB) of N Is 171
The least significant Byte (LSB) of N Is 129
The Binary representation of the MSB Is 10101011
The Binary representation of the LSB Is 10000001
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- [HiByte](#)
- [LoWord](#)
- [HiWord](#)

Returns the file position where the last file read/write was performed

Syntax

```
Declare Function LOC ( ByVal filenum As Long ) As LongInt
```

Usage

```
result = LOC( filenum )
```

Parameters

filenum

The file number of an open file.

Return Value

The file position where the last read/write was performed.

Description

Returns the position where the last file read/write was performed.

The position is indicated in records:

In files opened FOR RANDOM the record length specified when file w

In text files (FOR INPUT|OUTPUT|APPEND, a record length of 128 b

In files opened for BINARY a 1 byte record length is used.

In FreeBASIC the file position is 1 based, the first record of a file is re

When used with a serial device, **Loc** returns the number of bytes waiti
device's input buffer.

Example

```
Dim b As String
```

```
If Open Com ("com1:9600,n,8,1,cs,rs,ds,bin" For Bi
  Print "unable to open serial port"
  End
End If

Print "Sending command: AT"

Print #1, "AT" + Chr(13, 10);

Sleep 500,1

Print "Response:"

While( LOC(1) > 0 )
  b = Input(LOC(1), 1)
  Print b;
Wend

Close #1
```

Differences from QB

- !!WRITEME!! ?

See also

- LOF
- EOF
- Seek (Function)
- Open


```
errno = Err
Print "Error "; errno      ' just print the error
Sleep
End Sub
```

Differences from QB

- The LOCAL clause comes from PDS 7.1. QB 4.5 does not allow handling.

See also

- [On Error](#)

Locate



Sets the current cursor position

Syntax

```
Declare Function Locate( row As Long = 0, column As Long = 0,  
state As Long = -1, start As Long = 0, stop As Long = 0 ) As Long
```

Usage

```
Locate [row], [column], [state]  
  
result = Locate( [row], [column], [state] )  
new_column = LoByte( result )  
new_row = HiByte( result )  
new_state = HiWord( result )
```

Parameters

row
the 1-based vertical character position in the console.

column
the 1-based horizontal character position in the console.

state
the state of the cursor. 0 is off, 1 is on (console-mode only).

start
Ignored. Allowed for *-lang qb* dialect compatibility only.

stop
Ignored. Allowed for *-lang qb* dialect compatibility only.

Return Value

Returns a 32 bit **Long** containing the current cursor position and state. The **Low Byte Of The Low Word** contains the column, the **High Byte Of The Low Word** contains the row, and the **High Word** contains the cursor state.

If any of the *row*, *column* or *state* parameters were just set by the call to **Locate**, then the return value will reflect these new values, not the previous ones. If any of the parameters were omitted in the call to **Locate**, then the return value will reflect the current values, which are

the same as before the call to **Locate**.

Description

Sets the text cursor in both graphics and console modes.

Example

```
Locate 10  
Print "Current line:"; CsrLin
```

```
' ' Text cursor + mouse tracking  
Dim As Integer x = 0, y = 0, dx, dy  
  
Cls  
Locate , , 1  
  
While Inkey <> Chr(27)  
    GetMouse dx, dy  
    If( dx <> x Or dy <> y ) Then  
        Locate y+1, x+1: Print " ";  
        x = dx  
        y = dy  
        Locate 1, 1: Print x, y, ""  
        Locate y+1, x+1: Print "X";  
    End If  
Wend
```

Differences from QB

- The *start* and *stop* arguments have no effect in FreeBASIC.

See also

- CsrLin
- Pos
- (Print | ?)

Lock



Restricts read/write access to a file or portion of a file

Syntax

```
Lock #filenum, record  
Lock #filenum, start To end
```

Parameters

filenum

The file number used to **open** the file.

record

The record (**Random** files) to lock.

start

The first byte position (**Binary** files) to lock from.

end

The last byte position (**Binary** files) to lock to.

Description

Lock temporarily restricts access by other threads or programs to a file usually to allow safe writing to it.

After modifying the data, an **unlock** with the same parameters as the **lock**

Note: This command does not always work, neither as documented. It appears to be broken at the moment.

Example

```
' ' e.g. locking a file, reading 100 bytes, and un  
' ' To run, make sure there exists a file called 'f  
' ' in the current directory that is at least 100 b  
  
Dim array(1 To 100) As Integer  
Dim f As Integer, i As Integer  
f = FreeFile  
Open "file.ext" For Binary As #f
```

```
Lock #f, 1 To 100
For i = 1 To 100
    Get #f, i, array(i)
Next
Unlock #f, 1 To 100
Close #f
```

Differences from QB

- Currently, FB cannot implicitly lock the entire file
- In **Random** mode, FB cannot lock a range of records

See also

- **Open**
- **Unlock**
- **ScreenLock**

Returns the length of an open disk file

Syntax

```
Declare Function LOF ( ByVal filenum As Long ) As LongInt
```

Usage

```
result = LOF( filenum )
```

Parameters

filenum

The file number of an open disk file.

Return Value

The length in bytes of an open disk file.

Description

Returns the length, in bytes, of a file opened previously with **open** using the given *filenum*.

With **open com** it returns the length of the data pending to be read in the receive buffer.

Example

```
Dim f As Integer
f = FreeFile
Open "file.ext" For Binary As #f
Print LOF(f)
Close #f
```

Differences from QB

- None

See also

- [LOC](#)
- [EOF](#)
- [Open](#)

Log



Returns the natural logarithm of a given number

Syntax

```
Declare Function Log cdecl ( ByVal number As Double ) As Double
```

Usage

```
result = Log( number )
```

Parameters

number

The number to calculate the natural log.

Return Value

Returns the logarithm with the base e (also known as the natural logarithm)

Description

There can be some confusion with this notation given that in mathematics the natural logarithm is usually denoted **LN**, while the logarithm of base 10 is often denoted **LOG**. In programming languages, **LOG** is used to denote the natural logarithm. The argument must be a valid numeric expression greater than zero. If *number* is zero, FreeBASIC returns a string representing infinity, printing like "-Inf". If *number* is less than zero, **Log** returns a string representing NaN, printing like "NaN" or "IND", exact text is platform dependent. If *number* is not a number, an error is returned.

Example

```
'Find the logarithm of any base
Function LogBaseX (ByVal Number As Double, ByVal BaseX As Double) As Double
    LogBaseX = Log( Number ) / Log( BaseX )
'For reference: 1/log(10)=0.43429448
End Function
```

```
Print "The log base 10 of 20 is: "; LogBaseX ( 20 ,  
Print "The log base 2 of 16 is: "; LogBaseX ( 16 ,  
  
Sleep
```

The output would look like:

```
The log base 10 of 20 is: 1.301029995663981  
The log base 2 of 16 is: 4
```

Differences from QB

- None

See also

- [Exp](#)

Long



Standard data type: 32-bit signed integer

Syntax

```
Dim variable As Long
```

Description

32-bit signed whole-number data type. Can hold values from -2147483648 to 2147483647. Corresponds to a signed DWORD.

Example

```
Dim x As Long = &H80000000
Dim y As Long = &H7FFFFFFF
Print "Long Range = "; x; " to "; y
```

Output:

```
Long Range = -2147483648 to 2147483647
```

See also

- Integer
- LongInt
- ULong

LongInt



Standard data type: 64 bit signed

Syntax

```
Dim variable As LongInt
```

Description

A 64-bit signed whole-number data type. Can hold values from -9 223 036 854 775 808 to 9 223 372 036 854 775 807. Corresponds to a signed QWORD.

Example

```
Dim x As LongInt = &H8000000000000000
Dim y As LongInt = &H7FFFFFFFFFFFFFFF
Print "LongInt Range = "; x; " to "; y
```

Output:

```
LongInt Range = -9223372036854775808 to 9223372036854775807
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__LongInt`.

Differences from QB

- New to FreeBASIC

See also

- `ULongInt`
- `CLngInt`

Control flow statement for looping.

Syntax

```
Do  
[ statement block ]  
Loop [ { Until | While } condition ]
```

See also

- [Do...Loop](#)

Gets the lowest 16bit word of the operand.

Syntax

```
#define LoWord( expr ) (Cast(UInteger, expr) And &h0000FFFF;)
```

Usage

```
result = LoWord( expr )
```

Parameters

expr

A numeric expression, converted to an **UInteger** value.

Return Value

Returns the value of the low word of *expr*.

Description

This macro converts the numeric expression *expr* to an **UInteger** value representing the value of its least-significant (low) 16bit word.

Example

```
Dim N As UInteger

'Note there are 32 bits
N = &b10000000000000001111111111111111

Print "N is
Print "The binary representation of N is
Print "The most significant word (MSW) of N is
Print "The least significant word (LSW) of N is
Print "The binary representation of the MSW is
Print "The binary representation of the LSW is
```

Sleep

The output would look like:

```
N Is 2147614719
The Binary representation of N Is 10000000000000001111
The most significant word (MSW) of N Is 32769
The least significant word (LSW) of N Is 65535
The Binary representation of the MSW Is 1000000000000001
The Binary representation of the LSW Is 1111111111111111
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- [LoByte](#)
- [HiByte](#)
- [HiWord](#)

Lpos



Returns the number of characters sent to the printer port in the last **LPrint**

Syntax

Declare Function **Lpos** (**ByVal printer As Long**) **As Long**

Usage

```
result = LPOS(printer)
```

Parameters

printer

Either 0, 1, 2 or 3. Represents the printer port (LPT#)

Return Value

Returns the number of characters sent.

Description

Used to determine, from the last **LPrint**, how many characters were s

Example

```
' compile with -lang fblite or qb
#lang "fblite"
Dim test As String = "LPrint Example test"
Print "Sending '" + test + "' to LPT1 (default)"
LPrint test
Print "LPT1 last recieved " + Str(LPOS(1)) + " cha
Print "String sent was " + Str(Len(test)) + " char
Sleep
```

Differences from QB

- None

See also

- [LPrint](#)

Writes text to the default printer.

Syntax

```
LPrint [ Using formatstring, ] [expressionlist] [(, | ;)] ...
```

Parameters

formatstring

String specifying the output format.

expressionlist

List of variables to output according to the specified format.

Description

Prints *expressionlist* to the printer attached to the parallel port LPT1, or if it does not exist, to the default printer. To print to a printer different from the default one, use **Open Lpt**.

The **Using** clause formats *expressionlist* according to *formatstring*. Except an UDT, any data type can be passed to **LPrint** *expressionlist*, expressions do not need to be first converted to strings.

Using a comma (,) as separator or in the end of the *expressionlist* will place the cursor in the next column (every 14 characters), using a semi-colon (;) won't move the cursor. If neither of them are used in the end of the *expressionlist*, then a new-line will be printed.

Some printers will not print at all until a **chr(12)** (End of Page) character is printed.

Internally, FreeBASIC uses the special file number -1 for printing using **LPrint**. This file number may be safely closed using **close -1**. The next use of **LPrint** will automatically reopen it as needed.

Example

```
' ' Compile with -lang fblite or qb
#lang "fblite"

' ' new-line
LPrint "Hello World!"

' ' no new-line
LPrint "Hello"; "World"; "!";

LPrint

' ' column separator
LPrint "Hello!", "World!"

' ' end of page
LPrint Chr$(12)
```

Differences from QB

- None

Dialect Differences

- `LPrint` is not supported in the *-lang fb* dialect. In this dialect the printer must be properly opened with `open Lpt` and `Print # mu` be used to print.

See also

- `Open Lpt`
- `(Print | ?)`
- `(Print | ?) #`
- `Write`

Left-justifies a string

Syntax

```
Declare Sub LSet ( ByRef dst As String, ByRef src As Const String )
Declare Sub LSet ( ByVal dst As WString Ptr, ByVal src As Const WString Ptr )
```

Usage

```
LSet dst, src
LSet dst_udt, src_udt
```

Parameters

dst
String **string** to receive the data.

src
Source **string** to get the data.

dst_udt
User defined **Type** to receive the data.

src_udt
User defined **Type** to copy the data from.

Description

Lset left justifies text into the string buffer *dst*, filling the left part of the string with *src* and the right part with spaces. The string buffer size is not modified.

If text is too long for the string buffer size, **Lset** truncates characters from the right.

For compatibility with QBasic, **Lset** can also copy a user defined type variable into another one. The copy is made byte for byte, without any care for fields or alignment. It's up to the programmer to take care for the validity of the result.

Example

```
Dim buffer As String
buffer = Space(10)
LSet buffer, "91.5"
Print "-[" & buffer & "]"
```

```
Type mytype1
    x As Integer
    y As Integer
End Type

Type mytype2
    z As Integer
End Type

Dim a As mytype1 , b As mytype2
b.z = 1234

LSet a, b
Print a.x
```

Differences from QB

- In QB, the syntax was `Lset dst = src`. That syntax is also supported by FB.

See also

- RSet
- Space
- Put (File I/O)
- MKD
- MKI

- MKL
- MKS

Removes surrounding substrings or characters on the left side of a string.

Syntax

```
Declare Function LTrim ( ByRef str As Const String, [ Any ] ByRef  
trimset As Const String = " " ) As String  
Declare Function LTrim ( ByRef str As Const WString, [ Any ]  
ByRef trimset As Const WString = WStr(" ") ) As WString
```

Usage

```
result = LTrim[$]( str [, [ Any ] trimset ] )
```

Parameters

str
The source string.

trimset
The substring to trim.

Return Value

Returns the trimmed string.

Description

This procedure trims surrounding characters from the left (beginning) of a source string. Substrings matching *trimset* will be trimmed if specified, otherwise spaces (**ASCII** code 32) are trimmed.

If the **Any** keyword is used, any character matching a character in *trimset* will be trimmed.

All comparisons are case-sensitive.

Example

```
Dim s1 As String = " 101 Things to do."
```

```

Print "" + LTrim(s1) + ""
Print "" + LTrim(s1, " 01") + ""
Print "" + LTrim(s1, Any " 01") + ""

Dim s2 As String = "BaaBaaBAA Test Pattern"
Print "" + LTrim(s2, "Baa") + ""
Print "" + LTrim(s2, Any "BaA") + ""

```

will produce the output:

```

'101 Things to do.'
' 101 Things to do.'
'Things to do.'
'BAA Test Pattern'
' Test Pattern'

```

Platform Differences

- DOS version/target of FreeBASIC does not support the wide-character version of `LTrim`.

Dialect Differences

- The string type suffix "\$" is obligatory in the *-lang qb* dialect.
- The string type suffix "\$" is optional in the *-lang fblite* and *-lang fb* dialects.

Differences from QB

- QB does not support specifying a *trimset* string or the `ANY` clause.

See also

- `RTrim`
- `Trim`

Mid (Statement)



Overwrites a substring of a string with another

Syntax

```
Declare Sub Mid ( ByRef text As String, ByVal start As Integer,  
ByVal length As Integer, ByRef expression As Const String )  
Declare Sub Mid ( ByVal text As WString Ptr, ByVal start As  
Integer, ByVal length As Integer, ByVal expression As Const  
WString Ptr )
```

Usage

```
Mid( text, start ) = expression  
Or  
Mid( text, start, length ) = expression
```

Parameters

text

The string to work with.

start

The start position in *text* of the substring to overwrite. The first character starts at position 1.

length

The number of characters to overwrite.

Description

Copies a maximum of *length* characters of *expression* into *text*, starting at *start*.

If *length* is not specified, all of *expression* is copied. The size of the string *text* is unchanged; if *expression* is too big, as much of it is copied up to the end of *text*.

`Mid` can also be used as a function to return part of another string. See [Mid \(Function\)](#).

Example

```
Dim text As String

text = "abc 123"
Print text 'displays "abc 123"

' replace part of text with another string
Mid(text, 5, 3) = "456"
Print text 'displays "abc 456"
```

Differences from QB

- None

See also

- **Mid (Function)**

Mid (Function)



Returns a substring of a string

Syntax

Declare Function Mid (**ByRef** *str* as **Const String**, **ByVal** *start* as integer) as **String**

Declare Function Mid (**ByVal** *str* as **Const WString Ptr**, **ByVal** *start* as integer) as **WString**

Declare Function Mid (**ByRef** *str* as **Const String**, **ByVal** *start* as integer, **ByVal** *n* as integer) as **String**

Declare Function Mid (**ByVal** *str* as **Const WString Ptr**, **ByVal** *start* as integer, **ByVal** *n* as integer) as **WString**

Usage

```
result = Mid[$]( str, start [, n ] )
```

Parameters

str

The source string.

start

The start position in *str* of the substring. The first character starts at position 1.

n

The substring length, in characters.

Description

Returns a substring starting from *start* in *str*. If *str* is empty then the null string ("") is returned. If *start* <= 0 then the null string ("") is returned.

In the first form of `Mid`, all of the remaining characters are returned. In the second form, if *n* < 0 or *n* >= `len(str)` then all of the remaining characters are returned.

Example

```
Print Mid("abcdefg", 3, 2)
Print Mid("abcdefg", 3)
Print Mid("abcdefg", 2, 1)
```

will produce the output:

```
cd
cdefg
b
```

A Unicode example:

Wiki: code rendered this way to allow display of the Unicode characters.

```
dim text as wstring * 20
text = "Привет, мир!"
print mid(text, 6, 4) ' displays "т, м"
```

Platform Differences

- DOS does not support the wide-character string versions of `Mid`.

Dialect Differences

- The string type suffix "\$" is obligatory in the *-lang qb* dialect.
- The string type suffix "\$" is optional in the *-lang fblite* and *-lang fbc* dialects.

Differences from QB

- QB does not support Unicode.

See also

- InStr
- Mid (Statement)
- Left
- Right
- Asc

Minute



Gets the minute of the hour from a **Date Serial**

Syntax

```
Declare Function Minute ( ByVal date_serial As Double ) As Long
```

Usage

```
#include "vbcompat.bi"  
result = Minute( date_serial )
```

Parameters

date_serial
the date serial

Return Value

Returns the minute from a variable containing a date in **Date Serial** fo

Description

The compiler will not recognize this function unless `vbcompat.bi` is inc

Example

```
#include "vbcompat.bi"  
  
Dim ds As Double = DateSerial(2005, 11, 28) + Time  
Print Format(ds, "yyyy/mm/dd hh:mm:ss "); Minute(c
```

Differences from QB

- Did not exist in QB. This function appeared in PDS and VBDO:

See also

- **Date Serials**

Does a binary copy from a **Double** variable to a **String**, setting its length to 8 bytes

Syntax

```
Declare Function MKD ( ByVal number As Double ) As String
```

Usage

```
result = MKD[$]( number )
```

Parameters

number

A **Double** variable to binary copy to a **String**.

Return Value

Returns a **String** with a binary copy of the **Double**.

Description

Does a binary copy from a **Double** variable to a **String**, setting its length to 8 bytes. The resulting string can be read back to a **Double** by **CVD**.

This function is useful to write numeric values to buffers without using a **Type** definition.

Example

```
Dim n As Double, e As String
n = 1.2345
e = MKD(n)
Print n, CVD(e)
```

Dialect Differences

- The string type suffix "\$" is obligatory in the *-lang qb* dialect.
- The string type suffix "\$" is optional in the *-lang fblite* and *-lan fb* dialects.

Differences from QB

- None

See also

- MKI
- MKL
- MKS
- CVD
- CVI
- CVL
- CVS

MkDir



Makes a folder/directory on the local file system

Syntax

```
Declare Function MkDir ( ByRef folder As Const String ) As Long
```

Usage

```
result = MkDir( folder )
```

Parameters

folder

The folder/directory to be created.

Return Value

Returns zero (0) on success, and negative one (-1) on failure.

Description

Creates a folder on the local file system.

Example

```
Dim pathname As String = "foo\bar\baz"  
Dim result As Integer = MkDir( pathname )  
  
If 0 <> result Then Print "error: unable to create"
```

Platform Differences

- Linux requires the *filename* case matches the real name of the
- Path separators in Linux are forward slashes / . Windows uses backward \ slashes.

Differences from QB

- None

See also

- [Shell](#)
- [ChDir](#)
- [RmDir](#)

Does a binary copy from an integer variable to a **String** of the same length as the size of the input variable

Syntax

```
Declare Function MKI ( ByVal number As Integer ) As String
Declare Function MKI<bits> ( ByVal number As Integer<bits> ) As String
```

Usage

```
result = MKI[$]( number )
result = MKI[$]<bits>( number )
```

Parameters

number

A **Integer** or **Integer<bits>** variable to binary copy to a **String**.

Return Value

Returns a **String** containing a binary copy of *number*.

Description

Does a binary copy from an **Integer** or **Integer<bits>** variable to a **String**, setting its length to the number of bytes in the type. The resulting string can be read back to an integer type using **CVI** or **CVI<bits>**.

This function is useful to write numeric values to buffers without using a **Type** definition.

MKI supports an optional *<bits>* parameter before the argument. If *bits* is 16, **MKShort** will be called instead; if *bits* is 32, **MKL** will be called; if *bits* is 64, **MKLongInt** will be called. The length of the return value and the required *number* argument type will depend on which function is called. See each function's page for more information.

Example

```
Dim a As Integer, b As String
a=4534
b=MKI(a)
Print a, CVI(b)
```

Dialect Differences

- In the *-lang qb* dialect, **MKI** returns a 2-byte-string, since a QB integer is only 16 bits.
- The string type suffix "\$" is obligatory in the *-lang qb* dialect.
- The string type suffix "\$" is optional in the *-lang fblite* and *-lang fb* dialects.
- QB did not support a *<bits>* parameter.

See also

- **CVI**
- **MKShort**
- **MKL**
- **MKLongInt**
- **Integer**

Does a binary copy from a **Long** variable to a **String**, setting its length to bytes

Syntax

```
Declare Function MKL ( ByVal number As Long ) As String
```

Usage

```
result = MKL( number )
```

Parameters

number

A **Long** variable to binary copy to a **String**.

Return Value

Returns a **string** with a binary copy of the **Long**.

Description

Does a binary copy from a **Long** variable to a **String**, setting its length to 4 bytes. The resulting string can be read back to a **Long** by **CVL**.

This function is useful to write numeric values to buffers without using a **Type** definition.

Example

```
Dim a As Long, b As String
a = 4534
b = MKL(a)
Print a, CVL(b)
Sleep
```

Dialect Differences

- The string type suffix "\$" is obligatory in the *-lang qb* dialect.
- The string type suffix "\$" is optional in the *-lang fblite* and *-lan fb* dialects.

Differences from QB

- None

See also

- MKD
- MKI
- MKS
- CVD
- CVI
- CVL
- CVS

MKLongInt



Does a binary copy from a **LongInt** variable to a **String**, setting its length to 8 bytes

Syntax

```
Declare Function MKLongInt ( ByVal number As LongInt ) As String
```

Usage

```
result = MKLongInt[$]( number )
```

Parameters

number

A **LongInt** variable to binary copy to a **String**.

Return Value

Returns a **String** with a binary copy of the **LongInt**.

Description

Does a binary copy from a **LongInt** variable to a string, setting its length to 8 bytes. The resulting string can be read back to a longint by **CVLongInt**

This function is useful to write numeric values to buffers without using a **Type** definition.

Example

```
Dim a As LongInt, b As String
a = 4534
b = MKLongInt(a)
Print a, CVLongInt(b)
Sleep
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__Mklongint`.

Differences from QB

- New to FreeBASIC

See also

- `CVLongInt`

Does a binary copy from a **Single** variable to a **String**, setting its length to 4 bytes

Syntax

```
Declare Function MKS ( ByVal number As Single ) As String
```

Usage

```
result = MKS[$]( number )
```

Parameters

number

A **Single** variable to binary copy to a **String**.

Return Value

Returns a **string** with a binary copy of the **Single**.

Description

Does a binary copy from a **Single** variable to a **String**, setting its length to 4 bytes. The resulting string can be read back to a **Single** by **CVS**.

This function is useful to write numeric values to buffers without using a **Type** definition.

Example

```
Dim n As Single, e As String
n = 1.2345
e = MKS(n)
Print n, CVS(e)
```

Dialect Differences

- The string type suffix "\$" is obligatory in the *-lang qb* dialect.
- The string type suffix "\$" is optional in the *-lang fblite* and *-lan fb* dialects.

Differences from QB

- None

See also

- MKI
- MKL
- MKS
- CVD
- CVI
- CVL
- CVS

MKShort



Does a binary copy from a **short** variable to a **string**, setting its length to 2 bytes

Syntax

```
Declare Function MKShort ( ByVal number As Short ) As String
```

Usage

```
result = MKShort[$](number)
```

Parameters

number

A **short** variable to binary copy to a **string**.

Return Value

Returns a **string** with a binary copy of the **short**.

Description

Does a binary copy from a **SHORT** variable to a **string**, setting its length to 2 bytes. The resulting string can be read back to a **short** by **CVShort**

This function is useful to write numeric values to buffers without using a **Type** definition.

Example

```
Dim a As Short, b As String
a = 4534
b = MKShort(a)
Print a, CVShort(b)
Sleep
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__Mkshort`.

Differences from QB

- In QBasic this function is called `MKI`.

See also

- `CVShort`

Operator Mod (Modulus)



Finds the remainder from a division operation

Syntax

```
Declare Operator Mod ( ByRef lhs As Integer, ByRef rhs As Integer ) As Integer
```

Usage

```
result = lhs Mod rhs
```

Parameters

lhs

The left-hand side dividend expression.

rhs

The right-hand side divisor expression.

Return Value

Returns the remainder of a division operation.

Description

operator Mod (Modulus) divides two **Integer** expressions and returns the remainder. Numeric values are converted to **Integer** by rounding up or down.

Neither of the operands are modified in any way.

This operator can be overloaded for user-defined types.

Example

```
Print 47 Mod 7  
Print 5.6 Mod 2.1  
Print 5.1 Mod 2.8
```

Output:

```
5
0
2
```

This is because:

- 47 divided by 7 gives a remainder of 5
- 5.6 is rounded to 6 while 2.1 is rounded to 2. This makes the problem $6 \text{ MOD } 2$ which means 6 divided by 2 which gives a remainder of 0
- 5.1 is rounded to 5 while 2.8 is rounded to 3. This makes the problem $5 \text{ MOD } 3$ which means 5 divided by 3 which gives a remainder of 2

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- None

See also

- **Mathematical Functions**

Month



Gets the month of the year from a **Date Serial**

Syntax

```
Declare Function Month ( ByVal date_serial As Double ) As Long
```

Usage

```
#include "vbcompat.bi"  
result = Month( date_serial )
```

Parameters

date_serial
the date

Return Value

Returns the month number from a variable containing a date in **Date Serial**

The month values are in the range 1-12 being 1 for January and 12 for December

Description

The compiler will not recognize this function unless `vbcompat.bi` is included

Example

```
#include "vbcompat.bi"  
  
Dim a As Double = DateSerial(2005,11,28) + TimeSerial(12,59,59)  
Print Format(a, "yyyy/mm/dd hh:mm:ss "); Month(a)
```

Differences from QB

- Did not exist in QB. This function appeared in PDS and VBDO:

See also

- **Date Serials**

MonthName



Gets the name of a month from its integral representation

Syntax

```
Declare Function MonthName ( ByVal month As Long, ByVal abbrevia
```

Usage

```
#include "vbcompat.bi"  
result = MonthName( month_number [, abbreviate ] )
```

Parameters

month

the number of the month of the year - 1:January through 12:Decembe

abbreviate

flag to indicate that name should be abbreviated

Return Value

Returns the local operating system language month name from *month*

Description

If *abbreviate* is true, the month name abbreviation is returned. If omitted, the full name is returned.

The compiler will not recognize this function unless `vbcompat.bi` or `data`

Example

```
#include "vbcompat.bi"  
  
Dim ds As Double = DateSerial(2005, 11, 28) + TimeSerial(12, 59, 59)  
Print Format(ds, "yyyy/mm/dd hh:mm:ss "); MonthName(ds, True)
```

Differences from QB

- Did not exist in QB. This function appeared in Visual Basic.

See also

- **Date Serials**

MultiKey



Detects the status of keys by keyboard scancode.

Syntax

```
Declare Function MultiKey ( ByVal scancode As Long ) As Long
```

Usage

```
result = MultiKey(scancode)
```

Parameters

scancode

The **scan code** of the key to check.

Return Value

Returns -1 if the key for the specified **scan code** is pressed, otherwise

Description

MultiKey is a function which will detect the status of any key, determine if it is pressed, and return -1 if the key is pressed, otherwise it will return 0. The keyboard use **MultiKey**; that is, pressed keys will be stored and subsequently returned by **Inkey**. This means you have to empty **Inkey** manually when you finish using **MultiKey** method:

```
While Inkey <> "": Wend ' loop until the Inkey buffer is empty
```

Keeping **Inkey** to work while you use **MultiKey** allows more flexibility a **chr(255)+"k"** combo returned on window close button click, if a window is closed. For a list of accepted scancodes, see **DOS key** statement. For a list of accepted scancodes, see **DOS key** statement. **MultiKey** should always work in graphics mode, as long as the screen depends on the platform the program is run on though, and cannot be

Example

```

#include "fbgfx.bi"
#if __FB_LANG__ = "fb"
Using FB ' ' Scan code constants are stored in the
#endif

Dim As Integer x, y

ScreenRes 640, 480

Color 2, 15

x = 320: y = 240
Do
    ' Check arrow keys and update the (x, y) position
    If MultiKey(SC_LEFT ) And x > 0 Then x = x - 1
    If MultiKey(SC_RIGHT) And x < 639 Then x = x + 1
    If MultiKey(SC_UP   ) And y > 0 Then y = y - 1
    If MultiKey(SC_DOWN ) And y < 479 Then y = y + 1

    ' Lock the page while we work on it
    ScreenLock
        ' Clear the screen and draw a circle at the current position
        Cls
        Circle(x, y), 30, , , , ,F
    ScreenUnlock

    Sleep 15, 1

    ' Run loop until user presses Escape
Loop Until MultiKey(SC_ESCAPE)

' Clear Inkey buffer
While Inkey <> "": Wend

Print "Press CTRL and H to exit..."

Do

```

```
Sleep 25
```

```
'' Stay in loop until user holds down CTRL and  
If MultiKey(SC_CONTROL) And MultiKey(SC_H) The  
Loop
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- [Keyboard scancodes](#)
- [GetMouse](#)
- [GetJoystick](#)
- [Screen \(Graphics\)](#)
- [Inkey](#)

MutexCreate



Creates a mutex used for synchronizing the execution of threads

Syntax

Declare Function `MutexCreate () As Any Ptr`

Usage

`result = MutexCreate`

Return Value

The `Any Ptr` handle of the mutex created, or the null pointer (0) on fail

Description

Mutexes, short for "Mutually Exclusive", are a way of synchronizing sh (or a local static variable used by a single thread called multiple times `MutexLock` with that mutex (including the main thread executing main p

Mutexcreate creates a mutex, returning a handle which is to be referre
Mutexcreate should be destroyed when no longer needed or before th

A mutex is a lock that guarantees three things:

1. Atomicity - Locking a mutex is an atomic operation, meaning that th no other thread succeeded in locking this mutex at the same time.
2. Singularity - If a thread managed to lock a mutex, it is assured that the lock.
3. Non-Busy Wait - If a thread attempts to lock a thread that was locke any CPU resources) until the lock is freed by the second thread. At th locked by it.

Example

See also the `ThreadCreate` examples.

```
'Visual example of mutual exclusion between 2 thre
```

```

'the "user-defined thread" computes the points coordinates
'and the "main thread" plots the points.
'
'Principle of mutual exclusion
'
'      Thread#A                                XOR
'
'.....
'MutexLock(mut)                                Mut
'  Do_something#A_with_exclusion                D
'MutexUnlock(mut)                              Mut
'.....
'
'Behavior:
'- The first point must be pre-calculated.
'- Nothing prevents that a same calculated point could
'  (depends on execution times of the loops between
'- Nothing prevents that a calculated point could
'  (same remark on the loop times).
'
'If you comment out the lines containing "MutexLock"
'(inside "user-defined thread" or/and "main thread")
'there will be no longer mutual exclusion between
'and many points will not be plotted on circle (due to
'
'-----
Type ThreadUDT
  Dim handle As Any Ptr
  Dim sync As Any Ptr
  Dim quit As Byte
  Declare Static Sub Thread (ByVal As Any Ptr)
  Dim procedure As Sub (ByVal As Any Ptr)
  Dim p As Any Ptr
  Const false As Byte = 0
  Const true As Byte = Not false
End Type

Static Sub ThreadUDT.Thread (ByVal param As Any Ptr)
  Dim tp As ThreadUDT Ptr = param
  Do

```

```

        Static As Integer I
        MutexLock(tp->sync)
        tp->procedure(tp->p)
        I += 1
        Locate 30, 38
        Print I;
        MutexUnlock(tp->sync)
        Sleep 5
    Loop Until tp->quit = tp->>true
End Sub

```

```

Type Point2D

```

```

    Dim x As Integer
    Dim y As Integer

```

```

End Type

```

```

Const x0 As Integer = 640 / 2
Const y0 As Integer = 480 / 2
Const r0 As Integer = 200
Const pi As Single = 4 * Atn(1)

```

```

Sub PointOnCircle (ByVal p As Any Ptr)

```

```

    Dim pp As Point2D Ptr = p
    Dim teta As Single = 2 * pi * Rnd
    pp->x = x0 + r0 * Cos(teta)
    Sleep 5
    pp->y = y0 + r0 * Sin(teta)

```

'To increas

```

End Sub

```

```

Screen 12
Locate 30, 2
Print "<any_key> : exit";
Locate 30, 27
Print "calculated:";
Locate 30, 54
Print "plotted:";

```

```

Dim Pptr As Point2D Ptr = New Point2D
PointOnCircle(Pptr) ' Computatio

Dim Tptr As ThreadUDT Ptr = New ThreadUDT
Tptr->sync = MutexCreate
Tptr->procedure = @PointOnCircle
Tptr->p = Pptr
Tptr->handle = ThreadCreate(@ThreadUDT.Thread, Tptr

Do
    Static As Integer I
    Sleep 5
    MutexLock(Tptr->sync) 'Mutex (Lock) for main
    PSet (Pptr->x, Pptr->y) 'Plotting one point
    I += 1
    Locate 30, 62
    Print I;
    MutexUnlock(Tptr->sync) 'Mutex (Unlock) for ma
Loop Until Inkey <> ""

Tptr->quit = Tptr->>true
ThreadWait(Tptr->handle)
MutexDestroy(Tptr->sync)
Delete Tptr
Delete Pptr

Sleep

```

See also the similar `condcreate` example

Dialect Differences

- Threading is not allowed in the *-lang qb* dialect.

Platform Differences

- The DOS version of FreeBASIC does not allow for threads, as

- In Linux the threads are always started in the order they are cr

Differences from QB

- New to FreeBASIC

See also

- **MutexDestroy**
- **MutexLock**
- **MutexUnlock**
- **ThreadCreate**
- **ThreadWait**

MutexDestroy



Destroys a mutex

Syntax

```
Declare Sub MutexDestroy ( ByVal id As Any Ptr )
```

Usage

```
MutexDestroy( id )
```

Parameters

id

The **Any Ptr** handle of the mutex to be destroyed.

Description

Mutexdestroy discards a mutex created by **MutexCreate**. This call should be executed after any threads using the mutex are no longer in use.

See **MutexCreate** for more general information on mutexes.

Example

See the examples in **MutexCreate** and also **ThreadCreate**.

Dialect Differences

- Threading is not allowed in the **-lang qb** dialect.

Platform Differences

- The DOS version of FreeBASIC does not allow for threads, as the OS does not support them.
- In Linux the threads are always started in the order they are created, this can't be assumed in Win32. It's an OS, not a FreeBASIC issue.

Differences from QB

- New to FreeBASIC

See also

- **MutexCreate**
- **MutexLock**
- **MutexUnlock**
- **ThreadCreate**
- **ThreadWait**

MutexLock



Acquires a mutex

Syntax

```
Declare Sub MutexLock ( ByVal id As Any Ptr )
```

Usage

```
MutexLock( id )
```

Parameters

id

The **Any Ptr** handle of the mutex to be locked.

Description

Mutexlock halts any other threads using a mutex "handle", generated by `MutexCreate` and unlocked with `MutexUnlock`.

See `MutexCreate` for more general information on mutexes.

Example

See also the examples in `MutexCreate` and also `ThreadCreate`.

```
'Example of mutual exclusion for synchronization by
'by using 2 Mutexes only (by self lock and mutual
'The Producer works one time, then the Consumer works
'
'Principle of synchronisation by mutual exclusion
'(initial condition: mut#A and mut#B locked)
'
'          Thread#A          XORs
'Do_something#A_with_exclusion      MutexLock(
'MutexUnlock(mut#A)                  Do_somet
'.....                             MutexUnloc
```

```

'MutexLock(mut#B) .....

'-----

Dim Shared produced As Any Ptr
Dim Shared consumed As Any Ptr
Dim consumer_id As Any Ptr
Dim producer_id As Any Ptr

Sub consumer ( ByVal param As Any Ptr )
    For i As Integer = 0 To 9
        MutexLock produced
        Print , ",consumer gets:" ; i
        MutexUnlock consumed
        Sleep 5
    Next i
End Sub

Sub producer ( ByVal param As Any Ptr )
    For i As Integer = 0 To 9
        Print "Producer puts:" ; i;
        MutexUnlock produced
        MutexLock consumed
        Sleep 5
    Next i
End Sub

produced = MutexCreate
consumed = MutexCreate
If ( produced = 0 ) Or ( consumed = 0 ) Then
    Print "Error creating mutexes! Exiting..."
    Sleep
End
End If

MutexLock produced
MutexLock consumed

```

```
consumer_id = ThreadCreate ( @ consumer )
producer_id = ThreadCreate ( @ producer )
If ( producer_id = 0 ) Or ( consumer_id = 0 ) Then
    Print "Error creating threads! Exiting..."
    Sleep
End
End If

ThreadWait consumer_id
ThreadWait producer_id

MutexDestroy consumed
MutexDestroy produced

Sleep
```

Dialect Differences

- Threading is not allowed in the *-lang qb* dialect.

Platform Differences

- The DOS version of FreeBASIC does not allow for threads, as
- In Linux the threads are always started in the order they are created. Win32. It's an OS, not a FreeBASIC issue.

Differences from QB

- New to FreeBASIC

See also

- [MutexCreate](#)
- [MutexDestroy](#)
- [MutexUnlock](#)

- ThreadCreate
- ThreadWait

MutexUnlock



Releases a mutex lock

Syntax

```
Declare Sub MutexUnlock ( ByVal id As Any Ptr )
```

Usage

```
MutexUnlock( id )
```

Parameters

id

The **Any Ptr** handle of the mutex to be unlocked.

Description

Mutexunlock releases a mutex "handle" created by **MutexCreate**, and locked with **MutexLock**. This allows other threads sharing the mutex to continue execution.

See **MutexCreate** for more general information on mutexes.

Example

See the examples in **MutexCreate** and also **ThreadCreate**.

Dialect Differences

- Threading is not allowed in the **-lang qb** dialect.

Platform Differences

- The DOS version of FreeBASIC does not allow for threads, as the OS does not support them.
- In Linux the threads are always started in the order they are created, this can't be assumed in Win32. It's an OS, not a FreeBASIC issue.

Differences from QB

- New to FreeBASIC

See also

- [MutexCreate](#)
- [MutexDestroy](#)
- [MutexLock](#)
- [ThreadCreate](#)
- [ThreadWait](#)

Write functions without prolog/epilog code

Syntax

```
{Sub | Function} identifier Naked [calling_convention] ( param_list  
asm_statements  
End {Sub | Function}
```

Parameters

identifier - name of the procedure.

calling_convention - calling convention of the procedure - can be **cdecl**

asm_statements - the code in the procedure body. The code for handling these can change, depending on the **calling convention**.

param_list - parameters to be passed to the procedure.

data_type - the **data type** of the function.

Description

naked allows the programmer to write procedures without the compiler adding any unnecessary overhead.

Example

```
' ' Naked cdecl function  
Function subtract_c Naked cdecl _ ' ' parameters  
  ( _  
    ByVal a As Long, _  
    ByVal b As Long _ ' ' parameter push  
  ) As Long  
  
  Asm  
    mov eax, dword Ptr [esp+4] ' ' eax = a  
    Sub eax, dword Ptr [esp+8] ' ' eax -= b  
    ret ' ' return result  
  End Asm
```

End Function

```
Print subtract_c( 5, 1 ) '' 5 - 1
```

```
''-----  
  
'' Naked stdcall function  
Function subtract_s Naked stdcall _ '' parameters  
                                _ '' called proc  
                                _ '' (appendir  
  
    ( _  
        ByVal a As Long, _  
        ByVal b As Long _ '' parameter push  
    ) As Long  
  
    Asm  
        mov eax, dword Ptr [esp+4] '' eax = a  
        Sub eax, dword Ptr [esp+8] '' eax -= b  
        ret 8 '' return resu  
    End Asm
```

End Function

```
Print subtract_s( 5, 1 ) '' 5 - 1
```

```
''-----  
  
'' Naked pascal function  
Function subtract_p Naked pascal _ '' parameters  
                                _ '' called proc  
                                _ '' (appendir  
  
    ( _  
        ByVal a As Long, _ '' parameter push  
        ByVal b As Long _  
    ) As Long  
  
    Asm  
        mov eax, dword Ptr [esp+8] '' eax = a  
        Sub eax, dword Ptr [esp+4] '' eax -= b
```

```

        ret 8                                '' return result
    End Asm

End Function

Print subtract_p( 5, 1 ) '' 5 - 1

```

```

'' Naked cdecl function
'' plus ecx register preserved in asm block by cdecl
Function subtract_cp Naked cdecl _         '' parameter
( _
    ByVal a As Long, _
    ByVal b As Long _                     '' parameter
) As Long

    Asm
        push ebp                            '' push ebp
        mov ebp, esp                        '' ebp = esp
                                           '' => cdecl

        push ecx                            '' push ecx
        mov eax, dword Ptr [(ebp+4)+4]     '' eax = a
        mov ecx, dword Ptr [(ebp+8)+4]     '' ecx = b
        Sub eax, ecx                        '' eax -= ecx
        pop ecx                             '' pop ecx
        mov esp, ebp                        '' esp = ebp
        pop ebp                             '' pop ebp
                                           '' => cdecl

        ret                                 '' return
    End Asm

End Function

Print subtract_cp( 5, 1 ) '' 5 - 1

```

Platform Differences

- The default calling convention depends on the target platform,

Differences from QB

- New to FreeBASIC

See also

- `Asm`
- `Calling Conventions`
- `Function`
- `Sub`
- `cdecl`
- `pascal`
- `stdcall`

Name



Renames a file on disk

Syntax

```
Declare Function Name( ByRef oldname As Const String, ByRef newname As Long
```

Usage

```
result = Name( oldname, newname )
```

Parameters

oldname

Name of an existing file.

newname

New name of the file.

Return Value

Returns zero (0) on success and non-zero on failure.

Description

Renames a file or folder originally called *oldname* to *newname*.

The function is not guaranteed to succeed if a file/folder exists with the same name. It may succeed, overwriting the original, or it may fail. For greater control, [FileExists](#) could be used to test for an existing file, and [kill](#) could be used to delete an existing file.

Example

```
Dim OldName As String
Dim NewName As String
Dim result As Integer

OldName = "dsc001.jpg"
NewName = "landscape.jpg"
```

```
result = Name( OldName, NewName )
If 0 <> result Then
    Print "error renaming " & oldname & " to " & r
End If
```

Differences from QB

- In QB, NAME required AS rather than a comma between the o is because NAME was a language keyword rather than a funct

See also

- **Kill**
- **FileExists**

Declares a namespace block

Syntax

```
Namespace identifier [ Alias "aliasname" ]  
  statements  
End Namespace
```

Parameters

identifier

The name of the namespace (including nested names specifier).

aliasname

An alternate external name for the namespace.

Description

Namespaces allow to group entities like objects (predefined data-type declarations) under a name. This way the global scope can be divided

Whether or not explicitly declared a namespace in a source file, the current namespace, is present in every file.

Any identifier in the global namespace is available for use in a named namespace (declared inside a namespace).

Namespaces implicitly have public access and this is not modifiable.

A variable declared inside a namespace is always implicitly static and not specified (static and shared are optional, but this may improve code

Namespaces do not have any effect on the visibility of a define.

It is possible to define a namespace in two or more declarations.

Namespaces are commonly used in libraries where you don't want all symbols in the global namespace).

For example, if you used the "Forms" library, it might define the Point class for a certain purpose. This can be resolved by creating the namespace Forms for that purpose. Point.

To access duplicated symbols defined in the global namespace, use:

Example

```
Namespace Forms
  Type Point ' ' A 2D point
    As Integer x
    As Integer y
  End Type
  ' ' Since we are inside of the namespace, Point
  Sub AdjustPoint( ByRef pt As Point, ByVal newX
    pt.x = newX
    pt.y = newY
  End Sub
End Namespace

Type Point ' ' A 3D point
  As Integer x
  As Integer y
  As Integer z
End Type

Sub AdjustPoint( ByRef pt As Point, ByVal newX As
  pt.x = newX
  pt.y = newY
  pt.z = newZ
End Sub

Dim pt1 As Point
AdjustPoint( pt1, 1, 1, 1 )
Dim pt2 As Forms.Point
Forms.AdjustPoint( pt2, 1, 1 )
```

Namespaces are GCC C++ compatible, the following code aims to test

```
(cpp)
// mylib.cpp
// To compile:
```

```
//      g++ -c mylib.cpp -o mylib.o
//      ar rcs libmylib.a mylib.o

#include
#include

namespace mylib
{
    int test()
    {
        return 123;
    }
}
```

```
' ' test.bas

Extern "c++" Lib "mylib"
    Namespace mylib Alias "mylib"
        Declare Function test() As Integer
    End Namespace
End Extern

Print mylib.test()
```

Dialect Differences

- Namespaces are not supported in the *-lang qb* dialect.

Differences from QB

- New to FreeBASIC

See also

- [Using \(Namespaces\)](#)

Next



Control flow statement to mark the end of a **For . . .Next** loop.

Syntax

```
Next [ identifier_list ]
```

Description

Indicates the end of a statement block associated with a matching **For** statement.

When **Next** is used on its own without an *identifier_list*, it closes the most recent **For** statement block.

identifier_list is optional and may be one or more variable names separated by commas. This form of the **Next** statement is retained for compatibility with QB. *identifier_list*, if given, must match the identifiers used in the associated **For** statements in reverse order, from inner to outer.

Example

```
For i As Integer = 1 To 10
  For j As Integer = 1 To 2
    ' ...
  Next
Next
```

```
For i As Integer = 1 To 10
  For j As Integer = 1 To 2
    ' ...
  Next j
Next i
```

```
For i As Integer = 1 To 10
For j As Integer = 1 To 2
    ' ...
Next j,i
```

Differences from QB

- **ByRef** arguments cannot be used as counters.

See also

- **For...Next**

Operator New



Operator to dynamically allocate memory and construct data of a specific

Syntax

```
Declare Operator New ( size As UInteger ) As Any Ptr  
Declare Operator new[] ( size As UInteger ) As Any Ptr
```

Usage

```
result = New datatype  
or  
result = New datatype ( initializers, ... )  
or  
result = New datatype[ count ]
```

Parameters

size
Number of bytes to allocate.

initializers
Initial value(s) for the variable.

datatype
Name of the data type to create.

count
Exact number of elements to allocate.

Return Value

A pointer of type **datatype** to the newly allocated data.

Description

The **new** operator dynamically allocates memory and constructs a specified number of elements. For types with constructors, an initial value can be given. For types without constructors, Types that have constructors can have their constructors called by **new**. Default values for those types will be set.

new[] is the array-version of the **new** operator and allocates enough memory for the specified number of elements. The default constructor for the type will be used to set the initial value.

Objects created with **New** must be freed with **Delete**. Memory allocated array-version of **Delete**. You cannot mix and match the different versic

Specifying an initial value of **Any**, as in **New datatype(Any)** will allocate data. This is only valid on data types that do not have constructors (ot syntax of simple memory allocation with pointer conversion, like **Cptr(** be substituted to the invalid use of **New...Any**).

Specifying an initial value of **Any**, as in **New datatype[count]{Any}** will allocate the data. This is only valid on data types that do not have constructors syntax of simple memory allocation with pointer conversion, like **Cptr(** **Sizeof(datatype))**), can be substituted to the invalid use of **New...Any**)

Example

```
Type Rational
  As Integer      numerator, denominator
End Type

Scope

  ' Create and initialize a "rational" and store it
  Dim p As Rational Ptr = New Rational(3, 4)

  Print p->numerator & "/" & p->denominator

  ' Destroy the rational and give its memory back
  Delete p

End Scope

Scope

  ' Allocate memory for 100 integers and store the pointer
  Dim p As Integer Ptr = New Integer[100]

  ' Assign some values to the integers in the array
```

```
For i As Integer = 0 To 99
    p[i] = i
Next

' Free the entire integer array.
Delete[] p

End Scope
```

Dialect Differences

- Only available in the *-lang fb* dialect.

Differences from QB

- New to FreeBASIC

See also

- [Delete](#)
- [Placement New](#)

Operator Placement New



Operator to construct an object at a specified memory address.

Syntax

```
result = New(address) datatype  
or  
result = New(address) datatype ( initializers, ... )  
or  
result = New(address) datatype[ count ]
```

Parameters

address

the location in memory to construct. the parenthesis are **not** optional.

initializers

Initial value(s) for the variable.

datatype

name of the data type to construct.

count

Number of elements to construct.

Return Value

A pointer of type *datatype* to the newly constructed data.

Description

The **Placement New** operator constructs a specified data type at the sp

For simple types, like integers, an initial value can be given. For types field. Types that have constructors can have their constructors called values for those types will be set.

Memory is **not** allocated when using the **Placement New** operator. Instead it is incorrect to call **Delete** on the address. The proper way is to only use the syntax as for a member method by using member access operator. See examples below for proper *placement new* usage.

Specifying an initial value of **Any**, as in **New**(*address*)*datatype*(**Any**) OR **New**(*address*)*datatype*[**Any**]

This is only valid on data types that do not have constructors (otherwise, like `Cptr(datatype Ptr, address)`, can be substituted to the

Example

```
' ' "placement new" example

Type Rational
  As Integer      numerator, denominator
  Declare Constructor ( ByVal n As Integer, ByVal d As Integer
  As String ratio = "/"
End Type

Constructor Rational ( ByVal n As Integer, ByVal d As Integer
  This.numerator = n
  This.denominator = d
End Constructor

Scope

  ' ' allocate some memory to construct as a Rational
  Dim As Any Ptr ap = CAllocate(Len(Rational))

  ' ' make the placement new call
  Dim As Rational Ptr r = New (ap) Rational( 3, 4)

  ' ' you can see, the addresses are the same, just different
  Print ap, r

  ' ' confirm all is okay
  Print r->numerator & r->ratio & r->denominator

  ' ' delete must not be used with placement new
  ' ' destroying must be done explicitly if a destructor is defined
  ' ' (in this example, the var-string member is not a pointer)
  r->Destructor( )

  ' ' we explicitly allocated, so we explicitly deallocate
```

```
Deallocate( ap )
```

```
End Scope
```

Dialect Differences

- Only available in the *-lang fb* dialect.

Differences from QB

- New to FreeBASIC

See also

- [Destructor](#)
- [New](#)

Resume Next



Error handling statement to resume execution after a jump to an error handler.

Syntax

`Resume Next`

Description

`Resume Next` is used in the traditional QB error handling mechanism with the line after the one that caused the error. Usually this is used to avoid an error handler.

`Resume Next` resets the `Err` value to 0.

Example

```
' ' Compile with -lang fblite or qb
#lang "fblite"
Dim As Single i, j
On Error Goto ErrorHandler

i = 0
j = 5
j = 1 / i ' this line causes a divide-by-zero error
Print "ending..."

End ' end the program so that execution does not fall through

ErrorHandler:
Resume Next ' execution jumps to 'Print "ending..."
```

Dialect Differences

- RESUME NEXT is not supported in the *-lang fb* dialect. Stater

```
If Open( "text" For Input As #1 ) <> 0 Then
  Print "Unable to open file"
End If
```

Differences from QB

- Must compile with *-ex* option

See also

- [Err](#)
- [Resume](#)
- [Error Handling](#)

Operator Not (Complement)



Returns the bitwise-not (complement) of a numeric value

Syntax

```
Declare Operator Not ( ByRef rhs As Byte ) As Integer
Declare Operator Not ( ByRef rhs As UByte ) As Integer
Declare Operator Not ( ByRef rhs As Single ) As Integer
Declare Operator Not ( ByRef rhs As Double ) As Integer

Declare Operator Not ( ByRef rhs As T ) As T
```

Usage

```
result = Not rhs
```

Parameters

rhs
The right-hand side expression.

T
Any numeric or boolean type.

Return Value

Returns the bitwise-complement of its operand.

Description

This operator returns the bitwise-complement of its operand, a logical bits set depending on the bits of the operand.
(for a boolean type, 'Not false' returns 'true' and 'Not true' returns 'false')

The truth table below demonstrates all combinations of a boolean-con

Rhs Bit	Result
0	1
1	0

This operator can be overloaded for user-defined types.

Example

```
' Using the NOT operator on a numeric value
```

```
Dim numeric_value As Byte  
numeric_value = 15 '00001111
```

```
'Result = -16 =      11110000  
Print Not numeric_value
```

```
' Using the NOT operator on conditional expression
```

```
Dim As UByte numeric_value1, numeric_value2
```

```
numeric_value1 = 15
```

```
numeric_value2 = 25
```

```
If Not numeric_value1 = 10 Then Print "Numeric_Val
```

```
If Not numeric_value2 = 25 Then Print "Numeric_Val
```

```
' This will output "Numeric_Value1 is not equal to
```

```
' the first IF statement is false.
```

```
' It will not output the result of the second IF s
```

```
' condition is true.
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- None

See also

- **Operator Truth Tables**

Gets the current system time as a **Date Serial**

Syntax

```
Declare Function Now ( ) As Double
```

Usage

```
#include "vbcompat.bi"  
result = Now
```

Return Value

Returns a date serial containing the system's date and time at execution time.

Description

As the time is the decimal part of a date serial, if the value of **Now** is saved to an integer, the time in it will be reset to 00:00:00

The compiler will not recognize this function unless `vbcompat.bi` is included.

Example

```
#include "vbcompat.bi"  
  
Dim a As Double = Now()  
  
Print Format(a, "yyyy/mm/dd hh:mm:ss")
```

Differences from QB

- Did not exist in QB. This function appeared in PDS and VBDO:

See also

- **Date Serials**

Built-in type providing run-time type information

Syntax

```
Type object
As fb_BaseVT Ptr vtable_ptr
Declare Constructor()
End Type
```

Usage

```
Type typename Extends object
End Type
```

```
Dim variable As object
```

Description

Object is a built-in type which provides run-time type information for a types derived from it using **Extends**, allowing them to be used with **operator Is**, and to support **Virtual** and **Abstract** methods.

Extending the built-in **Object** type allows to add an extra hidden vtable pointer field at the top of the **Type**. The vtable is used to dispatch **Virtual** and **Abstract** methods and to access information for run-time type identification used by **operator Is**.

Example

See the **operator Is** page, the **Virtual** and **Abstract** pages.

Dialect Differences

- Not available in the **-lang qb** dialect unless referenced with the alias **__object**.

Differences from QB

- New to FreeBASIC

See also

- **Extends**
- **Operator Is**
- **Virtual**
- **Abstract**

Converts a number to octal representation

Syntax

```
Declare Function Oct ( ByVal number As UByte ) As String
Declare Function Oct ( ByVal number As UShort ) As String
Declare Function Oct ( ByVal number As ULong ) As String
Declare Function Oct ( ByVal number As ULongInt ) As String
Declare Function Oct ( ByVal number As Const Any Ptr ) As String

Declare Function Oct ( ByVal number As UByte, ByVal digits As
Long ) As String
Declare Function Oct ( ByVal number As UShort, ByVal digits As
Long ) As String
Declare Function Oct ( ByVal number As ULong, ByVal digits As
Long ) As String
Declare Function Oct ( ByVal number As ULongInt, ByVal digits As
Long ) As String
Declare Function Oct ( ByVal number As Const Any Ptr, ByVal
digits As Long ) As String
```

Usage

```
result = Oct[$]( number [, digits ] )
```

Parameters

number

A number or expression evaluating to a number. A floating-point number will be converted to a **LongInt**.

digits

Desired number of digits in the returned string.

Return Value

A string containing the unsigned octal representation of *number*.

Description

Returns the unsigned octal string representation of *number*. Octal digits range from 0 to 7.

If you specify *digits* > 0, the result string will be exactly that length. It will be truncated or padded with zeros on the left, if necessary.

The length of the returned string will not be longer than the maximum number of digits required for the type of *number* (3 characters for **Byte**, 6 for **Short**, 11 for **Long**, and 22 for **LongInt**)

If you want to do the opposite, i.e. convert an octal string back into a number, the easiest way to do it is to prepend the string with "&o;", and convert it to an integer type, using a function like **cInt**, similarly to a normal numeric string. E.g. **cInt("&o77;")**

Example

```
Print Oct(54321)
Print Oct(54321, 4)
Print Oct(54321, 8)
```

will produce the output:

```
152061
2061
00152061
```

Dialect Differences

- The string type suffix "\$" is obligatory in the **-lang qb** dialect.
- The string type suffix "\$" is optional in the **-lang fblite** and **-lang fb** dialects.

Differences from QB

- In QBASIC, there was no way to specify the number of digits returned.

- The size of the string returned was limited to 32 bits, or 11 octal digits.

See also

- `Bin`
- `Hex`
- `ValInt`
- `ValLng`

OffsetOf



Returns the offset of a field within a type.

Syntax

```
#define OffsetOf(typename, fieldname) CInt( @Cast( typename Ptr,  
0 )->fieldname )
```

Usage

```
result = offsetof( typename, fieldname )
```

Parameters

typename

Name of the type as defined using the **Type...End Type** statements.

fieldname

Name of the field as defined within the type (or within the base types for a derived type).

Description

For a non-derived type, **offsetof** will return the location *fieldname* as offset in bytes from the beginning of *typename*.

For a derived type, **offsetof** will return the location *fieldname* as offset in bytes from the beginning of its highest base type.

Note: if a member of the base type is overridden by a new member, the offset of the old member cannot be accessed from the derived type.

Example

```
Type MyType  
  x As Single  
  y As Single  
  Union  
    b As Byte  
    i As Integer  
  End Union
```

End Type

```
Print "OffsetOf x = "; OffsetOf(MyType, x)
Print "OffsetOf y = "; OffsetOf(MyType, y)
Print "OffsetOf b = "; OffsetOf(MyType, b)
Print "OffsetOf i = "; OffsetOf(MyType, i)
```

Output

```
OffsetOf x = 0
OffsetOf y = 4
OffsetOf b = 8
OffsetOf i = 8
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__offsetof`.

Differences from QB

- New to FreeBASIC

See also

- `Type...End Type`
- `SizeOf`

On Error



Error handling statement to set the current error handler

Syntax

```
On [Local] Error Goto label
```

Parameters

label

Label to jump to when an error occurs

Description

On Error triggers a jump to an error handler when an error occurs. Such as built-in statements such as **open**, or when the **Error** statement is used.

Note: The error checking for built-in statements is only enabled if the **error** option of the **-e**, **-ex** or **-exx** options. **On Error** remains working with **Error** even if the **error** options are used.

On Local Error can be used to specify a local error handler inside a procedure. It provides specialized per-procedure error handling and will override the global **Local Error** handler. The handler must be in the main part of the module.

Remark: Presently, the **Local** clause is ignored by the compiler.

On Error Goto 0 deactivates the current error handler.

Example

```
' ' Compile with QB (-lang qb) dialect
'$lang: "qb"

On Error Goto errorhandler
Error 24 ' simulate an error
Print "this message will not be seen"
```

```
errorhandler:
Print "Error #"; Err; "!"
End
```

```
' ' compile as: fbc onerror.bas -ex

#lang "fblite"

Function hFileExists( filename As String ) As Integer
    Dim f As Integer

    hFileExists = 0

    On Local Error Goto exitfunction

    f = FreeFile
    Open filename For Input As #f

    Close #f

    hFileExists = -1

exitfunction:
    Exit Function
End Function

Print "File exists (0=false): "; hFileExists(

    On Error Goto errhandler
    Error 1234
    Print "back from resume next"
    End 0

errhandler:
    Print "error number: " + Str( Err ) + " at lin
```

Resume Next

Differences from QB

- QB has no LOCAL clause and requires the label to be in the m

See also

- **Error**
- **Local**
- **Err**
- **Runtime Error Codes**
- **Error Handling**

On...Gosub



Calls a label based on an expression

Syntax

```
On expression GoSub label1[, ...]
```

Description

Branches to different labels depending on the value of *expression*. An expression value of 1 will branch to the first label, a value of 2 to the second, etc. If the value of *expression* is zero (0) or greater than the number of items in the list, execution continues on the next statement following the **On...Gosub**.

This statement behaves exactly like **GoSub** and execution may return to the statement following the **On...Gosub** using **Return**.

It is recommended that the structured **Select Case** conditional statement be used instead of **On...Gosub**.

Example

```
' ' Compile with -lang qb
'$lang: "qb"

choice = 3
On choice GoSub labela, labelb, labelc
Print "Good bye."
End

labela:
Print "choice a"
Return

labelb:
```

```
Print "choice b"  
Return  
  
labelc:  
Print "choice c"  
Return
```

Dialect Differences

- Only available in the *-lang qb* and *-lang fblite* dialects.
- `on gosub` support is disabled by default in the *-lang fblite* unless the `option gosub` statement is used.

Differences from QB

- FreeBASIC does not generate a run-time error if *expression* is negative or greater than 255.

See also

- `Select Case`
- `On...Goto`
- `GoSub`
- `Return`
- `Option Gosub`

On...Goto



Jumps to a label based on an expression.

Syntax

```
On expression Goto label1[, ...]
```

Description

Branches to different labels depending on the value of *expression*. An expression value of 1 will branch to the first label, a value of 2 to the second, etc. If the value of *expression* is zero (0) or greater than the number of items in the list, execution continues on the next statement following the `On...Goto`.

It is recommended that the structured `Select Case` conditional statement be used instead of `On...Goto`.

Example

```
Dim choice As Integer

Input "Enter a number: ", choice

On choice Goto labela, labelb, labelc

labela:
Print "choice a"
End

labelb:
Print "choice b"
End

labelc:
Print "choice c"
End
```

Differences from QB

- FreeBASIC does not generate a run-time error if *expression* is negative or greater than 255.

See also

- [Select Case](#)
- [On...Gosub](#)
- [Goto](#)

Opens a disk file for reading or writing using file operations

Syntax

```
Open filename For Input [encoding_type] [lock_type] As [#]filenu
Open filename For Output [encoding_type] [lock_type] As [#]filen
Open filename For Append [encoding_type] [lock_type] As [#]filen

Open filename For Binary [access_type] [lock_type] As [#]filenum
Open filename For Random [access_type] [lock_type] As [#]filenum
```

Usage

```
result = Open( filename[,] For {Input|Output|Append}[,] As filen
or
result = Open( filename[,] For Binary[,] Access {Read|Write}[,] As filer
or
result = Open( filename[,] For Random[,] Access {Read|Write}[,] As fil
or
Open filename For {Input|Output|Append} As filenumber
or
Open filename For Binary Access {Read|Write} As filenumber
or
Open filename For Random Access {Read|Write} As filenumber [Len
```

Parameters

filename

A string value of the name of the disk file to open. Relative file paths are `curDir`).

encoding_type

The encoding to be used when reading or writing text, can be one of:

- Encoding "ascii" (ASCII encoding is used, default)
- Encoding "utf8" (8-bit Unicode encoding is used)
- Encoding "utf16" (16-bit Unicode encoding is used)
- Encoding "utf32" (32-bit Unicode encoding is used)

access_type

By default, the **Binary** and **Random** file modes allow both reading and writing but this can be changed by specifying an access type (see the description above).

For any file mode, access to the opened disk file can be restricted or granted by specifying a lock type (see the description for the *lock_type* parameter). Other threads of the current program can freely open the disk file (Shared), while other threads can read (Read) or write (Write). **Lock** and **Unlock** can be used to temporarily restrict access to parts of the file.

The error code returned by **open** can be checked using **Err** in the next line. You can also check directly the error code as an integer.

Example

```
' Create a string and fill it.
Dim buffer As String, f As Integer
buffer = "Hello World within a file."

' Find the first free file number.
f = FreeFile

' Open the file "file.ext" for binary usage, using
Open "file.ext" For Binary As #f
If Err>0 Then Print "Error opening the file":End

' Place our string inside the file, using number "f"
Put #f, , buffer

' Close all open files.
Close

' End the program. (Check the file "file.ext" upon
End
```

```
' OPEN A COM PORT
```

```
Open Com "COM1:9600,N,8,1" As #1
If Err>0 Then Print "The port could not be opened."

'COM1, 9600 BAUD, NO PARITY BIT, EIGHT DATA BITS, 0
```

```
'function version of OPEN
If Open("file.ext" For Binary Access Read As #1) =

    Print "Successfully opened file"

    ' ...

    Close #1

Else

    Print "Error opening file"

End If
```

Platform Differences

- Linux requires the *filename* case matches the real name of the file is case insensitive.
- Path separators in Linux are forward slashes /. Windows uses back slashes \. DOS uses backward slashes \.
- On Windows, a file number used in a dynamic link library is not used in the main program. File numbers can not be passed or returned from a DLL and an executable.
- If you try to open a directory on Linux, the **open** command will succeed.

Differences from QB

- Using MS-DOS device names to open streams or hardware devices is only supported in the *-lang qb* dialect; for other modes FreeBASIC's new command syntax is used.

Open Com, Open Cons, Open Err, Open Pipe, Open Lpt, Open Scrn.

- **open** can be called as a function that returns an error code.

Dialect Differences

- The **-lang qb** dialect supports the old GW-BASIC-style syntax of `[length]` with `mode_string` being "I" for input, "O" for output, "A" binary.

See also

- **Err (and a list of error codes)**
- **Close**
- **FreeFile**
- **Open Cons, Open Err, Open Pipe, Open Lpt, Open Com, Open Scrn**

Open Com



Opens a serial port for input and output

Syntax

```
Declare Function Open Com ( byref options As String, As filenum  
As Long ) As Long
```

Usage

```
result = Open Com( options[,] As[#] filenum )
```

Parameters

options

A **String** containing options used in controlling the port.

filenum

The file number to bind to the port.

Return Value

Returns zero (0) on success and a non-zero error code otherwise.

Description

This command opens a serial port of the PC, allowing to send and receive data by using the normal file commands as **Print #**, **Input #**, **Get #**, ...

The main parameter is a **string** that describes, at the very least, which communications port to open. It has the format:

```
"Comn: [ baudrate ][ , [ parity ][ , [ data_bits ][ , [ stop_bit  
][ , [ extended_options ]]]]"
```

where,

n

Com port to open. "1", "2", "3", "4", etc. Some platforms will support more serial ports depending on how the operating system is configured. Where *n* is not given, "COM:" will map to "COM1:", except o

Linux where "COM:" maps to "/dev/modem"

baudrate

"300" (default), "1200", ..., etc.

parity

"N" (none), "E" (even, default), "O" (odd), "S" (space), "M" (mark), "PE" (QB-quirk: checked, even parity)

data_bits

"5", "6", "7" (default) or "8".

stop_bits

"1", "1.5" or "2". (*default value depends on baud rate and data bits, see table below*)

Condition	Default number of stop bits
baud rate <= 110 and data bits = 5	1.5
baud rate <= 110 and data bits >= 6	2
baud rate > 110	1

extended_options

Miscellaneous options. (*See table below*)

Option	Action
'CSn'	Set the CTS duration (in ms) (n>=0), 0 = turn off, default = 1000
'DSn'	Set the DSR duration (in ms) (n>=0), 0 = turn off, default = 1000
'CDn'	Set the Carrier Detect duration (in ms) (n>=0), 0 = turn off
'OPn'	Set the 'Open Timeout' (in ms) (n>=0), 0 = turn off
'TBn'	Set the 'Transmit Buffer' size (n>=0), 0 = default, depends on platform
'RBn'	Set the 'Receive Buffer' size (n>=0), 0 = default, depends on platform
'RS'	Suppress RTS detection
'LF'	Communicate in ASCII mode (add LF to every CR) - Win32 doesn't support this one
'ASC'	same as 'LF'
'BIN'	The opposite of LF and it'll always work
'PE'	Enable 'Parity' check
'DT'	Keep DTR enabled after CLOSE
'FE'	Discard invalid character on error
'ME'	Ignore all errors
'IRn'	IRQ number for COM (only supported (?) on DOS)

All items except for the COM port are optional. The order of *baudrate*, *parity*, *data_bits*, *stop_bits* is fixed. Any skipped fixed item (*baudrate*, etc...) must be empty.

Example

```
Open Com "COM1:9600,N,,2" As 1
```

Opens COM1 with 9600 baud, no parity, 7 data bits and 2 stop bits.

```
Open Com "COM1:115200" As 1
```

Opens COM1 with 115200 baud, "even" parity, 7 data bits and 1 stop bits.

Platform Differences

- On the Windows platform "COM:" maps to "COM1:"
- On the Linux platform

"COM:" maps to "/dev/modem"

"COM1:" maps to "/dev/ttyS0"

"COM2:" maps to "/dev/ttyS1", etc

"/dev/xyz:" maps to "/dev/xyz", etc

- The DOS serial driver is experimental and can access COM ports 1 to 4

It uses the following base io and IRQ's as default:

COM1 - &h3f8; - IRQ4

COM2 - &h2f8; - IRQ3

COM3 - &h3e8; - IRQ4

COM4 - &h2e8; - IRQ3

Since fbc-0.18.4, an alternate IRQ can be specified using the the "IRn" protocol option where *n* is 3 through 7.

Currently not supported: IRQ's on the slave PIC, alternate base I/O

addresses, Timeouts and most errors as detected in QB, hardware flow control, FIFO's.

"COM:" maps to "COM1:"

Dialect Differences

- In the *-lang qb* dialect the old syntax OPEN "COMx:..." is supported.

Differences from QB

- In QB the syntax was OPEN "COMx:[baudrate] [,parity, [data_bits, [stop_bits, [extended_options]]]" FOR INPUT|OUTPUT|RANDOM AS [#] n
- In QB, only "COM1:" and "COM2:" are supported. In FreeBASIC, any correctly configured serial port may be used.

See also

- [Open](#)

Open Cons



Opens the console's standard input (*stdin*) or output (*stdout*) streams for use in file operations.

Syntax

```
Open Cons As [#]filename  
Open Cons For Input As [#]filename  
Open Cons For Output As [#]filename
```

Usage

```
result = Open Cons( [For {Input|Output}[,]] As filename )  
(or using the QB-like syntax,)
```

```
Open Cons [For {Input|Output}] As filename
```

Parameters

filename

An available *file number* to bind to the *stdin* or *stdout* stream, which can be found with **FreeFile**.

Return Value

In the first usage, **open cons** returns zero (0) on success and a non-zero error code otherwise.

Description

open cons opens the console's *stdin* or *stdout* streams for reading or writing. A *file number* is bound to the stream, which is used in subsequent file operations, such as **Input #**. An available *file number* can be retrieved with **FreeFile**.

The **Input file mode** opens the *stdin* stream for reading file operations, such as **Line Input #**, while the **Output file mode** opens the *stdout* stream for writing file operations, such as **Print #**. The **Output file mode** is the default if not specified.

The *stdin* and *stdout* streams are the ones used when the calling process' input or output is redirected (piped) by OS commands, or when it is opened with **Open Pipe**.

To open both the *stdin* and *stdout* streams for file operations, a process must use multiple *file numbers*.

Runtime errors:

open cons throws one of the following **runtime errors**:

(1) Illegal function call

- *filenumber* was not free at the time. use **FreeFile** to ensure that *filenumber* is free.

Example

```
Dim a As String

Open Cons For Input As #1
Open Cons For Output As #2

Print #2, "Please write something and press ENTER"
Line Input #1, a
Print #2, "You wrote : ";a

Close
Sleep
```

Differences from QB

- In QB the syntax was OPEN "CON:" FOR INPUT|OUTPUT AS [#]
filenum

See also

- **Open**

- **Open Scrn**
- **Open Err**
- **FreeFile**

Open Err



Opens both the standard input (*stdin*) and standard error (*stderr*) stream file operations.

Syntax

```
Open Err [for mode] As [#]filenum As Long
```

Usage

```
Open Err [for mode] as [#]filenum  
or  
result = Open Err( [for mode[,]] as [#]filenum )
```

Parameters

mode
Ignored.
filenum
An unused file number.

Return Value

Zero is returned if **open Err** completed successfully, otherwise a non-z value is returned to indicate failure.

Description

This command opens *stdin* to read from and *stderr* to write to the console allowing read and write operations with normal file commands.

stderr is an output stream different from *stdout* allowing error messages be redirected separately from the main console output.

The normal console commands, such as **color** and **locate**, do not work in this mode, because they do not accept a file number.

The [For Input|Output] *mode* is allowed for compatibility, but is ignored.

Runtime errors:

`open Err` throws one of the following **runtime errors**:

(1) *Illegal function call*

- *Filenumber* was not free at the time. use `FreeFile` to ensure that *filenumber* is free.

Example

```
Dim a As String
Open Err For Input As #1
Print #1, "Please write something and press ENTER"
Line Input #1, a
Print #1, "You wrote"; a
Close
Sleep
```

Differences from QB

- New to FreeBASIC

See also

- `Open`

Open Lpt



Open a printer device

Syntax

```
Open Lpt ["[LPT[x]:][Printer_Name][,TITLE=Doc_Title][,EMU=TTY]" ]
```

Usage

```
Open Lpt "LPT..." As [#]filenum  
or  
result = Open Lpt( "LPT..."[, ] As [#]filenum )
```

Parameters

x
Specifies a port number. If omitted, output is sent to the system printer.
Printer_Name
Name of printer to open. This parameter is ignored on DOS.
TITLE=Doc_Title
Title of the print job as seen by the printer spooler. This parameter is ignored on DOS.
EMU=TTY
Emulation of TTY output on a windows GDI printer, using driver text input/output.
For Input|Output
clause is allowed for compatibility, but it is ignored.
filenum
An unused file number to assign to the device.

Return Value

0 is returned if `open Lpt` completed successfully, otherwise a non-zero

Description

`open Lpt` opens a connection to a printer device. The connection is terminated when the printer device is closed.

Any printer attached to the system may be opened with `open Lpt`.

`open Lpt "LPT:" ...` will try to open the default printer on Windows architecture.

`LPrint` will automatically try to open the default printer on Windows and

Platform specific notes:

Windows

The argument `EMU=TTY` assumes printable ASCII or Unicode text, and `TAB`, `FF`, etc., for virtual print-head movement...even when the printer is omitted, the data must be sent in the printer's language (ESC/P, HP

Linux

A printer spooler available through `lp` must be installed to access print spoolers may work that are invoked through `lp`. Port are zero-based c

The data must be sent in the printer's language (ESC/P, HPGL, PostS

DOS

FreeBASIC does not support print spoolers on DOS. Printers must be

The data must be sent in the printer's language (ESC/P, HPGL, PostS

Example

```
' Send some text to the Windows printer on LPT1:,  
Open Lpt "LPT1:EMU=TTY" For Output As #1  
Print #1, "Testing!"  
Close
```

```
' Sends contents of text file test.txt to Windows  
Dim RptInput As String  
Dim PrintFileNum As Integer, RptFileFileNum As Int  
  
RptFileFileNum = FreeFile  
Open "test.txt" For Input As #RptFileFileNum  
  
PrintFileNum = FreeFile  
Open Lpt "LPT:ReceiptPrinter,TITLE=ReceiptWinTitle  
#PrintFileNum
```

```

While (EOF(RptFileFileNum) = 0)
    Line Input #RptFileFileNum, RptInput
    Print #PrintFileNum, RptInput
Wend

Close #PrintFileNum ' Interestingly, does not rec

Close #RptFileFileNum

Print "Press any key to end program..."
GetKey

End

```

```

'This simple program will print a PostScript file
Dim As UByte FFI, PPO
Dim As String temp

FFI = FreeFile()
Open "sample.ps" For Input Access Read As #FFI
PPO = FreeFile()
Open Lpt "LPT1:" For Output As #PPO
While (EOF(FFI) = 0)
Line Input #FFI, temp
Print #PPO, temp
Wend

Close #FFI
Close #PPO

Print "Printing Completed!"

```

Dialect Differences

- In the *-lang qb* dialect the old syntax is supported `OPEN "LPT:`

See also

- `Open`
- `LPrint`

Open Pipe



Opens an external process' standard input (*stdin*) or output (*stdout*) stream.

Syntax

```
Open Pipe shell_command For Input As [#]filenumber  
Open Pipe shell_command For Output As [#]filenumber  
Open Pipe shell_command For Binary access_type [#]filenumber
```

Usage

```
result = Open Pipe( command[,] For {Input|Output}[,] As filenumber  
or,  
result = Open Pipe( command[,] For Binary[,] access_type[,] As filenumber  
(or in the QB-like syntax,)  
Open Pipe filename For {Input|Output} As filenumber  
(or,)  
Open Pipe filename For Binary access_type As filenumber
```

Parameters

shell_command

The external process to execute in the operating system command shell in the current directory (see **curDir**). When opening a pipe for a process that requires an executable path, or its arguments, the entire pipe string should be nested in quotes. *access_type*

The type of read or write access requested by the calling process.

- **Access** {Read|Write} (either the *stdin* or *stdout* stream)

filenumber

An available file number to bind to the external process' *stdin* or *stdout* stream.

Return Value

In the first usage, **Open Pipe** returns zero (0) on success and a non-zero value on error.

Description

Open Pipe executes another process in the command shell and opens a stream for reading or writing. A *file number* is bound to the stream, which is used in subsequent **File** statements.

Input #. An available *filenumber* can be retrieved with **FreeFile**. If the error is thrown.

The **Input** and **Output file modes** open the external process' *stdin* and *stdout* text I/O, useful for reading or writing plain text. Characters, words or whole lines are read or written in text-mode file operations, such as **Line Input #** and **Print #**.

The **Binary file mode** opens the external process' *stdin* or *stdout* stream specified (see description of the *access_type* parameter above) - for reading or writing arbitrarily sized and interpreted raw data. Simple data type values, like integers, can be read from or written to the streams with binary-mode file operations. Bidirectional pipes are not supported by FB and must be implemented manually.

Runtime errors:

Open Pipe throws one of the following **runtime errors**:

(1) *Illegal function call*

- *filenumber* was not free at the time. use **FreeFile** to ensure it is free.

Example

```
' ' This example uses Open Pipe to run a shell command
#ifdef __FB_UNIX__
Const TEST_COMMAND = "ls *"
#else
Const TEST_COMMAND = "dir *.*"
#endif

Open Pipe TEST_COMMAND For Input As #1

Dim As String ln
Do Until EOF(1)
    Line Input #1, ln
    Print ln
Loop

Close #1
```

Platform Differences

- The **Binary file mode** is not supported on all platforms; **Open Pipe** the external process' *stdin* or *stdout* streams in binary mode.

Differences from QB

- New to FreeBASIC

See also

- [Shell](#)
- [Open](#)
- [Open Cons](#)
- [Open Err](#)
- [FreeFile](#)

Open Scrn



Opens the console directly for input and output as a file

Syntax

```
Open Scrn [for mode] As [#]filenum As Long
```

Usage

```
Open Scrn [for mode] as [#]filenum  
or  
result = Open Scrn( [for mode[,]] as [#]filenum )
```

Parameters

mode

Either **Input** or **Output**. If omitted, **Output** is assumed.

filenum

An unused file number.

Return Value

Zero (0) is returned if **open Err** completed successfully, otherwise a non-zero value is returned to indicate failure.

Description

This command opens the console for both input and output as a file, allowing to read/write from/to it with normal file commands.

This command may use direct access to the console for speed in some implementations, so it should not be used when the input / output is required to be redirected or piped with OS commands.

The normal console commands, such as **Color** and **Locate**, do not work in this mode, because they do not accept a file number.

The [For Input|Output] clause is allowed for compatibility, but is ignored.

filenum is an unused file number.

An unused file number can be found using **FreeFile**.

Runtime errors:

`open cons` throws one of the following **runtime errors**:

(1) *Illegal function call*

- *filenumber* was not free at the time. use **FreeFile** to ensure that *filenumber* is free.

Example

```
Dim a As String
Open Scrn For Input As #1
Print #1, "Please write something and press ENTER"
Line Input #1, a
Print #1, "You wrote";a
Close
Sleep
```

Differences from QB

- QB used `OPEN "SCRN:" ...`

See also

- `Open`
- `Open Cons`

Declares or defines an overloaded operator.

Syntax

```
{ Type | Class | Union | Enum } typename
Declare Operator Cast () [ ByRef ] As datatype
Declare Operator @ () [ ByRef ] As datatype Ptr
Declare Operator assignment_op ( [ ByRef | ByVal ] rhs As datatype)
Declare Operator [] ( index As datatype ) [ ByRef ] As datatype
Declare Operator New ( size As UInteger ) As Any Ptr
Declare Operator New[] ( size As UInteger ) As Any Ptr
Declare Operator Delete ( buf As Any Ptr )
Declare Operator Delete[] ( buf As Any Ptr )
End { Type | Class | Union | Enum }
```



```
{ Type | Class | Union } typename
Declare Operator For ()
Declare Operator For ( [ ByRef | ByVal ] stp As typename )
Declare Operator Step ()
Declare Operator Step ( [ ByRef | ByVal ] stp As typename )
Declare Operator Next ( [ ByRef | ByVal ] cond As typename ) As
Declare Operator Next ( [ ByRef | ByVal ] cond As typename, [ By
End { Type | Class | Union }
```



```
Declare Operator unary_op ( [ ByRef | ByVal ] rhs As datatype )
Declare Operator binary_op ( [ ByRef | ByVal ] lhs As datatype,
```



```
Operator typename.Cast () [ ByRef ] As datatype [ Export ]
Operator typename.@ () [ ByRef ] As datatype Ptr [ Export ]
Operator typename.assignment_op ( [ ByRef | ByVal ] rhs As datatype)
Operator [] ( index As datatype ) [ ByRef ] As datatype [ Export ]
Operator unary_op ( [ ByRef | ByVal ] rhs As datatype ) As datatype
Operator binary_op ( [ ByRef | ByVal ] lhs As datatype, [ ByRef
Operator typename.New ( size as uinteger ) As Any Ptr [ Export ]
Operator typename.New[] ( size As UInteger ) As Any Ptr [ Export ]
Operator typename.Delete ( buf As Any Ptr ) [ Export ]
Operator typename.Delete[] ( buf As Any Ptr ) [ Export ]
```

Parameters

typename

Name of the **Type**, **Class**, **Union**, Or **Enum**.

assignment_op

let += -= *= &= /= \= mod= shl= shr= and= or= xor= imp= eqv= ^=

```
unary_op
- not * -> abs sgn fix frac int exp log sin asin cos acos tan at
binary_op
+ - * & / \ mod shl shr and or xor imp eqv ^ = <> < > <= >=
```

Description

The built in operators like `=`, `+`, and `cast` have predefined behaviors w/ to do something other than predefined operations when at least one o data type.

Operators are just functions. The operator `'+'` has functionality like `Fun` See ***Operator Overloading*** for more information. Operators can be o `cast` Operator is the only operator (or function) that can be declared n usage, the compiler may decide which cast overload to call based on

Non-static operator members are declared inside the **Type** or **Class**. G (procedure bodies) must appear outside.

Let, **cast**, and other assignment operators must be declared inside th have a return data type same as the **Type** or **Class** they are declared in

Unary operators must be declared outside the **Type**, **Class**, or **Enum** and can be overloaded to return any valid data type, except for **Operator - Class** data type.

Binary operators must be declared outside the **Type**, **Class**, or **Enum** and can be overloaded with valid data types, except for relational operator

Let refers to the assignment operator, as in `LET a=b`. The **Let** keyword **fb** dialect. However, **Let()** can be used to assign the fields of a UDT t

See **For**, **Step**, and **Next** for more information on overloading the **For..**

New, **New[]**, **Delete**, and **Delete[]** operator members are always static, but allowed).

Example

```

'' operator1.bas

Type Vector2D
  As Single x, y

  '' Return a string containing the vector data.
  Declare Operator Cast() As String

  '' Multiply the vector by a scalar.
  Declare Operator *= ( ByVal rhs As Single )
End Type

'' Allow two vectors to be able to be added together
Declare Operator + ( ByRef lhs As Vector2D, ByRef rhs As Vector2D ) As Vector2D

'' Return the modulus (single) of the vector using Pythagoras
Declare Operator Abs ( ByRef rhs As Vector2D ) As Single

Operator Vector2D.cast ( ) As String
  Return "(" + Str(x) + ", " + Str(y) + ")"
End Operator

Operator Vector2D.*= ( ByVal rhs As Single )
  This.x *= rhs
  This.y *= rhs
End Operator

Operator + ( ByRef lhs As Vector2D, ByRef rhs As Vector2D ) As Vector2D
  Return Type<Vector2D>( lhs.x + rhs.x, lhs.y + rhs.y )
End Operator

Operator Abs ( ByRef rhs As Vector2D ) As Single
  Return Sqr( rhs.x * rhs.x + rhs.y * rhs.y )
End Operator

Dim a As Vector2D = Type<Vector2D>( 1.2, 3.4 )
Dim b As Vector2D = Type<Vector2D>( 8.9, 6.7 )
Dim c As Vector2D = Type<Vector2D>( 4.3, 5.6 )

```

```

Print "a = "; a, "abs(a) ="; Abs( a )
Print "b = "; b, "abs(b) ="; Abs( b )
Print "a + b = "; a + b, "abs(a+b) ="; Abs( a + b )
Print "c = "; c, "abs(c) ="; Abs( c )
Print "'c *= 3'"
c *= 3
Print "c = "; c, "abs(c) ="; Abs( c )

```

Aligned memory allocator:

- by using the overloaded member operators "New" and "ALIGN" bytes (256 bytes in this example),
- the real pointer of the allocated memory is saved just at

```

'' operator2.bas

Const ALIGN = 256

Type UDT
  Dim As Byte a(0 To 10 * 1024 * 1024 - 1) '' 10 m
  Declare Operator New (ByVal size As UInteger) As Any Ptr
  Declare Operator Delete (ByVal buffer As Any Ptr) As Any Ptr
  Declare Constructor ( )
  Declare Destructor ( )
End Type

Operator UDT.New (ByVal size As UInteger) As Any Ptr
  Print " Overloaded New operator, with parameter size = " & size
  Dim pOrig As Any Ptr = CAllocate(ALIGN-1 + SizeOf(UDT Ptr))
  Dim pMin As Any Ptr = pOrig + SizeOf(UDT Ptr)
  Dim p As Any Ptr = pMin + ALIGN-1 - (CULng(pMin) - CULng(pMin))
  Cast(Any Ptr Ptr, p)[-1] = pOrig
  Operator = p
  Print " real pointer = &h" & Hex(pOrig), "returning pointer = " & Hex(p)
End Operator

Operator UDT.Delete (ByVal buffer As Any Ptr)
  Print " Overloaded Delete operator, with parameter " & Hex(buffer)
End Operator

```

```

    Dim pOrig As Any Ptr = Cast(Any Ptr Ptr, buffer)
    Deallocate(pOrig)
    Print "  real pointer = &h" & Hex(pOrig)
End Operator

Constructor UDT ()
    Print "  Constructor, @This = &h" & Hex(@This)
End Constructor

Destructor UDT ()
    Print "  Destructor, @This = &h" & Hex(@This)
End Destructor

Print "'Dim As UDT Ptr p = New UDT'"
Dim As UDT Ptr p = New UDT

Print "  p = &h" & Hex(p)

Print "'Delete p'"
Delete p

```

Output example:

```

'Dim As UDT Ptr p = New UDT'
  Overloaded New operator, with parameter size = &hA00000;
  real pointer = &h420020;   return pointer = &h420100;
  Constructor, @This = &h420100;
  p = &h420100;
'Delete p'
  Destructor, @This = &h420100;
  Overloaded Delete operator, with parameter buffer = &h420100
  real pointer = &h420020;

```

Small use case of the operator "[]": simplest smart pointers for byte bu

```

' ' operator3.bas

' ' A smart pointer is an object which behaves like
' ' - This object is flexible as a pointer and has
' '   like constructor and destructor called automa

```

```

'' - Therefore, the destructor of the smart pointer
''   when this object goes out of scope, and it will
''
'' Example of simplest smart pointers for byte buffers
'' - Constructor and destructor allow to allocate,
'' - Pointer index operator allows to access buffers
'' - Copy-constructor and let-operator are just defined
''   in order to disallow copy construction and assignment

```

```

Type smartByteBuffer

```

```

  Public:

```

```

    Declare Constructor (ByVal size As UInteger =
    Declare Operator [] (ByVal index As UInteger)
    Declare Destructor ()

```

```

  Private:

```

```

    Declare Constructor (ByRef rhs As smartByteBuffer
    Declare Operator Let (ByRef rhs As smartByteBuffer
    Dim As Byte Ptr psbb

```

```

End Type

```

```

Constructor smartByteBuffer (ByVal size As UInteger)

```

```

  This.destructor()

```

```

  If size > 0 Then

```

```

    This.psbb = New Byte[size]

```

```

    Print "Byte buffer allocated"

```

```

  End If

```

```

End Constructor

```

```

Operator smartByteBuffer.[] (ByVal index As UInteger)

```

```

  Return This.psbb[index]

```

```

End Operator

```

```

Destructor smartByteBuffer ()

```

```

  If This.psbb > 0 Then

```

```

    Delete[] This.psbb

```

```

    This.psbb = 0

```

```

    Print "Byte buffer deallocated"

```

```

  End If

```

```

End Destructor

```

```
Scope
  Dim As smartByteBuffer sbb = smartByteBuffer(256)
  For I As Integer = 0 To 255
    sbb[I] = I - 128
  Next I
  Print
  For I As Integer = 0 To 255
    Print Using "#####"; sbb[I];
  Next I
  Print
End Scope
```

Dialect Differences

- Only available in the *-lang fb* dialect.

See also

- [Class](#)
- [Enum](#)
- [Type](#)

Option()



Specifies additional attributes and/or characteristics of symbols.

Syntax

```
option( "literal-text" )
```

Parameters

literal-text

The literal text specifying the option. See description.

Description

`option()` allows the programmer to specify additional attributes or characteristics of symbols. Enclosing the string into quotes and parentheses is required in the syntax. Unrecognized options are ignored.

`option` can also be used as a statement to specify other compile time **Compiler Switches**.

Individual options are explained below.

SSE

`option("SSE")` indicates that a floating point value (**Single** or **Double**) register function is stored in the `xmm0` register. `option("Sse")` is ignored unless compiled with the **-fpu SSE** command line option. This option may be placed immediately after the return type in a function declaration or function call. This option is an optimization only and not required to compile programs using the **SSE** command line option.

```
Declare Function ValueInXmm0 () As Double Option("SSE")
```

FPU

`option("FPU")` indicates that a floating point value (**Single** or **Double**) return type is stored in the `st(0)` register. This option may be used immediately in a function declaration or function definition.

```
Declare Function ValueInStZero () As Double Option
```

Differences from QB

- New to FreeBASIC

See also

- **Compiler Option: -fpu**
- **Compiler Switches**

Option Base



Specifies a default lower bound for array declarations

Syntax

```
Option Base base_subscript
```

Parameters

base_subscript
an numeric literal value

Description

option Base is a statement that sets the default lower bound for any for array declarations. This default remains in effect for the rest of the module if not overridden, and can be overridden by declaring arrays with an explicit lower bound in an **option Base** statement.

Note: initially, the default base is 0.

Example

```
' ' Compile with the "-lang qb" or "-  
lang fblite" compiler switches  
  
#lang "fblite"  
  
Dim foo(10) As Integer      ' declares an array with 11 elements  
  
Option Base 5  
  
Dim bar(15) As Integer     ' declares an array with 16 elements  
Dim baz(0 To 4) As Integer ' declares an array with 5 elements
```

Dialect Differences

- Only available in the *-lang fblite* and *-lang qb* dialects.
- In *-lang fb*, `option Base` is not allowed, and the default lower bound is 0.

Differences from QB

- QBASIC only supported values of 0 or 1 for *base_subscript*.
- In QBASIC the word `Base` was a reserved keyword, and couldn't be used as a variable name.
- Arrays must always be explicitly created in FreeBASIC. QBASIC would automatically create an array from *base_subscript* to 10 if one was used in code without a declaration.

See also

- [Dim](#)
- [ReDim](#)
- [LBound](#)

Option ByVal



Specifies parameters are to be passed by value by default in procedure declarations

Syntax

```
Option ByVal
```

Description

`Option ByVal` is a statement that sets the default passing convention for procedure parameters to *by value*, as if declared with `ByVal`. This default remains in effect for the rest of the module in which `Option ByVal` is used, and can be overridden by specifying `ByRef` in parameter lists.

Example

```
' ' compile with the "-  
lang fblite" compiler switch  
  
#lang "fblite"  
  
Sub TestDefaultByref( a As Integer )  
    ' ' change the value  
    a = a * 2  
End Sub  
  
Option ByVal  
  
Sub TestDefaultByval( a As Integer )  
    a = a * 2  
End Sub  
  
Dim a As Integer = 1  
  
Print "a = "; a
```

```
TestDefaultByref( a )
Print "After TestDefaultByref : a = "; a
Print

Print "a = "; a
TestDefaultByval( a )
Print "After TestDefaultByval : a = "; a
Print
```

Dialect Differences

- Only available in the *-lang fblite* and *-lang qb* dialects.

Differences from QB

- New to FreeBASIC

See also

- `__FB_OPTION_BYVAL__`

Option Dynamic



Specifies variable-length array declarations

Syntax

`Option Dynamic`

Description

`Option Dynamic` is a statement that specifies that any following array declarations are variable-length, whether they are declared with `const` subscript ranges or not. This remains in effect for the rest of the module which `Option Dynamic` is used, and can be overridden with `Option Static` equivalent to the `'$Dynamic` metacommand.

Example

```
' ' Compile with "-lang fblite" compiler switch
#lang "fblite"

Dim foo(99) As Integer      ' declares a fixed-
length array

Option Dynamic

Dim bar(99) As Integer     ' declares a variable-
length array
' ...
ReDim bar(199) As Integer  ' resize the array
```

Dialect Differences

- Only available in the *-lang fblite* and *-lang qb* dialects.

Differences from QB

- New to FreeBASIC

See also

- `__FB_OPTION_DYNAMIC__`
- `'$Dynamic`
- `'$Static`
- `Option Static`
- `Dim`
- `ReDim`

Option Escape



Specifies that string literals should be processed for C-like escape sequences.

Syntax

`Option Escape`

Description

`Option Escape` is a statement that causes string literals to be processed by default. Normally, escape sequences have no effect in string literals with the **! Operator (Escaped String Literal)**. This default remains in the module in which `Option Escape` is used, and can be overridden by the **! Operator (Non-Escaped String Literal)**.

See **Literals** in the **Programmer's Guide** to learn more about escape sequences.

Example

```
' ' Compile with the "-lang fblite" compiler switch
#lang "fblite"

Option Escape

Print "Warning \a\t The path is:\r\n c:\\Freebasic
Print $"This string doesn't have expanded escape s

#include "crt.bi"

Dim As Integer a = 2, b = 3
printf("%d * %d = %d\r\n", a, b, a * b)
```

Dialect Differences

- Only available in the **-lang fblite** and **-lang qb** dialects.

Differences from QB

- New to FreeBASIC

See also

- `__FB_OPTION_ESCAPE__`
- **Operator ! (Escaped String Literal)**
- **Operator \$ (Non-Escaped String Literal)**
- **Literals**

Option Explicit



Forces variables, objects and arrays to be declared before they are used.

Syntax

`Option Explicit`

Description

`Option Explicit` is a statement that forces any following variable, object or array declaration, with, for example, `Dim` or `Static`. This rule remains in effect in all code in which `Option Explicit` is used, and cannot be overridden.

Example

```
' ' Compile with the "-lang qb" or "-lang fblite" compiler
#lang "fblite"

Option Explicit

Dim a As Integer      ' 'a' must be declared
a = 1                 ' ..or this statement
```

Dialect Differences

- Only available in the *-lang fblite* and *-lang qb* dialects.

Differences from QB

- New to FreeBASIC

See also

- `__FB_OPTION_EXPLICIT__`

Option Gosub



Enables support for `GoSub` and `On Gosub`.

Syntax

```
Option Gosub
```

Description

`option gosub` enables support for `GoSub` and `Return` (from `gosub`).

Because `Return` could mean return-from-gosub or return-from-procedure, `option gosub` and `option nogosub` can be used to enable and disable `GoSub` support. When `GoSub` support is disabled, `Return` is then recognized as return-from-procedure.

Example

```
' ' Compile with the "-  
lang fblite" compiler switch  
  
#lang "fblite"  
  
' ' turn on gosub support  
Option GoSub  
  
GoSub there  
backagain:  
    Print "backagain"  
    End  
  
there:  
    Print "there"  
    Return
```

Dialect Differences

- Only available in the *-lang fblite* and *-lang qb* dialects.

Differences from QB

- New to FreeBASIC

See also

- [__Fb_Option_Gosub__](#)
- [Option Nogosub](#)
- [GoSub](#)
- [Return](#)

Option Nogosub



Disables support for `GoSub` and `On Gosub`.

Syntax

```
Option Nogosub
```

Description

`option Nogosub` disables support for `GoSub` and `Return` (from `gosub`).

Because `Return` could mean return-from-gosub or return-from-procedure, `option Gosub` and `option Nogosub` can be used to enable and disable `GoSub` support. When `GoSub` support is disabled, `Return` is then recognized as return-from-procedure.

Example

```
' ' Compile with the "-lang qb" compiler switch
'$lang: "qb"

' ' turn off gosub support
Option nogosub

Function foo() As Integer
    Return 1234
End Function

Print foo
```

Dialect Differences

- Only available in the *-lang fblite* and *-lang qb* dialects.

Differences from QB

- New to FreeBASIC

See also

- `__Fb_Option_Gosub__`
- `Option Gosub`
- `GoSub`
- `Return`

Option NoKeyword



"Undefines" a reserved keyword

Syntax

```
Option NoKeyword keyword
```

Parameters

keyword
the keyword to undefine

Description

`Option NoKeyword` is a statement that undefines a FreeBASIC reserved identifier for a variable, object, procedure or any other symbol. The the module in which `Option NoKeyword` is used.

Example

```
' ' Compile with the "-lang fblite" compiler switch
#lang "fblite"

Option NoKeyword Int           ' remove the keyword '
                               ' symbol table

Dim Int As Integer           ' declare a variable w
```

Dialect Differences

- Only available in the *-lang fblite* and *-lang qb* dialects.

Differences from QB

- New to FreeBASIC

See also

- `#undef`

Option Private



Specifies internal linkage by default for procedure declarations

Syntax

```
Option Private
```

Description

Option Private is a statement that gives any following procedure declarations internal linkage by default, as if declared with **Private**. This default remains in effect for the rest of the module in which **Option Private** is used, and can be overridden by declaring procedures with **Public**.

Example

```
' ' Compile with the "-  
lang fblite" compiler switch  
  
#lang "fblite"  
  
Sub ProcWithExternalLinkage()  
    '  
    ...  
End Sub  
  
Option Private  
  
Sub ProcWithInternalLinkage()  
    '  
    ...  
End Sub  
  
Public Sub AnotherProcWithExternalLinkage()  
    '  
    ...  
End Sub
```

Dialect Differences

- Only available in the *-lang fblite* and *-lang qb* dialects.

Differences from QB

- New to FreeBASIC

See also

- `__FB_OPTION_PRIVATE__`
- `Private`
- `Public`

Option Static



Reverts to default array declaration behavior

Syntax

```
Option Static
```

Description

`option static` is a statement that overrides the behavior of `option dynamic`. Arrays declared with constant subscript ranges are fixed-length. This is the default behavior for the rest of the module in which `option static` is used, and can be overridden by `option dynamic`. It is equivalent to the '\$Static metacommand.

Example

```
' ' Compile with the "-lang fblite" compiler switch
#lang "fblite"

Option Dynamic

Dim foo(100) As Integer      ' declares a variable-length array

Option Static

Dim bar(100) As Integer     ' declares a fixed-length array
```

Dialect Differences

- Only available in the *-lang fblite* and *-lang qb* dialects.

Differences from QB

- New to FreeBASIC

See also

- `'$Dynamic`
- `'$Static`
- `Dim`
- `Erase`
- `ReDim`
- `Option Dynamic`
- `Static`

Operator Or (Inclusive Disjunction)



Returns the bitwise-or (inclusive disjunction) of two numeric values

Syntax

```
Declare Operator Or ( ByRef lhs As T1, ByRef rhs As T2 ) As Ret
```

Usage

```
result = lhs Or rhs
```

Parameters

lhs

The left-hand side expression.

T1

Any numeric or boolean type.

rhs

The right-hand side expression.

T2

Any numeric or boolean type.

Ret

A numeric or boolean type (varies with *T1* and *T2*).

Return Value

Returns the bitwise-disjunction of the two operands.

Description

This operator returns the bitwise-disjunction of its operands, a logical depending on the bits of the operands (for conversion of a boolean to -1 integer value).

The truth table below demonstrates all combinations of a boolean-disj

Lhs Bit	Rhs Bit	Result
0	0	0
1	0	1

0	1	1
1	1	1

No short-circuiting is performed - both expressions are always evaluated.

The return type depends on the types of values passed. **Byte**, **UByte** and **Integer**. If the left and right-hand side types differ only in signedness, type (τ_1), otherwise, the larger of the two types is returned. Only if the return type is also **Boolean**.

This operator can be overloaded for user-defined types.

Example

```
' Using the OR operator on two numeric values
Dim As UByte numeric_value1, numeric_value2
numeric_value1 = 15 '00001111
numeric_value2 = 30 '00011110

'Result = 31 = 00011111
Print numeric_value1 Or numeric_value2
Sleep
```

```
' Using the OR operator on two conditional expressions
Dim As UByte numeric_value
numeric_value = 10

If numeric_value = 5 Or numeric_value = 10 Then Print "5 or 10"
Sleep

' This will output "Numeric_Value equals 5 or 10"
' while the first condition of the first IF statement is false
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- None

See also

- [OrElse](#)
- [Operator Truth Tables](#)

Or



Parameter to the **Put** graphics statement which uses a bit-wise **or** as the

Syntax

```
Put [ target, ] [ STEP ] ( x,y ), source [ ,( x1,y1 )-( x2,y2 )
```

Parameters

or
Required.

Description

The **or** method combines each source pixel with the corresponding destination pixel using the bit-wise **or** function. The result of this is output as the destination. This method works in all graphics modes. There is no mask color, although values of 0 (RGBA(0, 0, 0, 0) in full-color modes) will have no effect, the behavior of **or**.

In full-color modes, each component (red, green, blue and alpha) is kept as a set of bits, so the operation can be made to only affect some of the channels, making sure the all the values of the other channels are set to 0.

Example

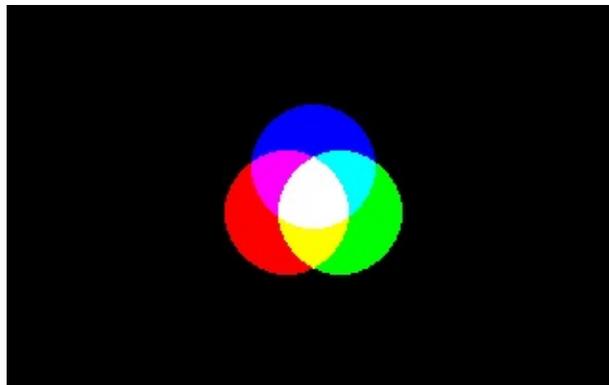
```
'open a graphics window
ScreenRes 320, 200, 16

'create 3 sprites containing red, green and blue
Const As Integer r = 32
Dim As Any Ptr cr, cg, cb
cr = ImageCreate(r * 2 + 1, r * 2 + 1, RGBA(0, 0,
cg = ImageCreate(r * 2 + 1, r * 2 + 1, RGBA(0, 0,
cb = ImageCreate(r * 2 + 1, r * 2 + 1, RGBA(0, 0,
Circle cr, (r, r), r, RGB(255, 0, 0), , , 1, f
Circle cg, (r, r), r, RGB(0, 255, 0), , , 1, f
Circle cb, (r, r), r, RGB(0, 0, 255), , , 1, f
```

```
'put the sprite at three different multiplier
'levels, overlapping each other in the middle
Put (146 - r, 108 - r), cr, Or
Put (174 - r, 108 - r), cg, Or
Put (160 - r, 84 - r), cb, Or

'free the memory used by the sprites
ImageDestroy cr
ImageDestroy cg
ImageDestroy cb

'pause the program before closing
Sleep
```



Differences from QB

- None

See also

- Or
- Put (Graphics)

Operator OrElse (Short Circuit Inclusive Disjunction)

Returns the short circuit-or (Inclusive Disjunction) of two numeric values

Syntax

```
Declare Operator OrElse ( ByRef lhs As T1, ByRef rhs As T2 ) As
```

Usage

```
result = lhs OrElse rhs
```

Parameters

lhs

The left-hand side expression.

T1

Any numeric or boolean type.

rhs

The right-hand side expression.

T2

Any numeric or boolean type.

Ret

A numeric or boolean type (varies with *T1* and *T2*).

Return Value

Returns the short circuit-or (inclusive disjunction) of the two operands

Description

This operator evaluates the left hand side expression. If the result is nonzero, then -1 (true) is immediately returned. If the result is zero the right hand side is evaluated, and the logical result from that is returned returning -1 (true) for a nonzero value or 0 (false) for zero.

(for conversion of a boolean to an integer, false or true boolean value becomes 0 or -1 integer value)

The truth table below demonstrates all combinations of a short circuit-operation, the '-' denotes that the operand is not evaluated.

Lhs Value	Rhs Value	Result
0	0	0
0	nonzero	-1
nonzero	-	-1

Short-circuiting is performed - only expressions needed to calculate the result are evaluated.

The return type is almost always an **Integer**, of the value 0 or -1, denoting false and true respectively. Except if the left and right-hand side types both **Boolean**, then the return type is also **Boolean**.

This operator cannot be overloaded for user-defined types.

Example

```
' Using the ORELSE operator on two numeric values
Dim As Integer numeric_value1, numeric_value2
numeric_value1 = 15
numeric_value2 = 30

'Result = -1
Print numeric_value1 OrElse numeric_value2
Sleep
```

Differences from QB

- This operator was not available in QB.

See also

- **AndAlso**
- **Or**
- **Operator Truth Tables**

Outputs a value to a hardware port.

Syntax

```
Declare Function Out ( ByVal port As UShort , ByVal data As UByte Long
```

Usage

```
out port,value
```

Parameters

port

Hardware port to write to.

data

Data value to write.

Description

This function sends *value* to *port* and returns immediately.

Example

```
'speakersound.bas
Sub Sound(ByVal freq As UInteger, dur As UInteger)
  Dim t As Double, f1 As Unsigned Short
  f1 = 1193181 \ freq
  Out &h61, Inp(&h61) Or 3
  Out &h43, &hb6
  Out &h42, LoByte(f1)
  Out &h42, HiByte(f1)
  t=Timer
  While ((Timer - t) * 1000) < dur
    Sleep 0,1
  Wend
  Out &h61, Inp(&h61) And &hfc
End Sub
```

```
Sound(523, 60) 'C5
Sound(587, 60) 'D5
Sound(659, 60) 'E5
Sound(698, 60) 'F5
Sound(784, 60) 'G5
Sound(880, 60) 'A5
Sound(988, 60) 'B5
Sound(1046, 60) 'C6
```

Platform Differences

- In the Windows and Linux versions three port numbers (&H3C7, &H3C8;, &H3C9;) are hooked by the graphics library when a game mode is in use to emulate QB's VGA palette handling. This use is deprecated; use **Palette** to retrieve and set palette colors.
- Using true port access in the Windows version requires the user to install a device driver for the present session. For that reason, Windows executables using hardware port access should be run as administrator each time the computer is restarted. Future runs don't require admin rights as they just use the already installed driver. The driver is only 3K in size and is embedded in the executable.

See also

- **Inp**
- **Wait**
- **Palette**

Specifies text file to be opened for output mode

Syntax

```
Open filename for Output [Encoding encoding_type] [Lock lock_typ]  
[#] filenum
```

Parameters

filename

file name to open for output

encoding_type

indicates encoding type for the file

lock_type

locking to be used while the file is open

filenum

unused file number to associate with the open file

Description

A file mode used with **open** to open a text file for writing.

This mode is used to write text with **Print #**, or comma separated values with **Write #**.

Text files can't be simultaneously read and written in FreeBASIC, so if both **Open** and **Print #** functions are required on the same file, it must be opened twice.

filename must be a string expression resulting in a legal file name in the OS, without wildcards. The file will be sought for in the present directory unless the *filename* contains a path. If the file does not exist, it is created. The file pointer is set at the first character of the file.

Encoding_type indicates the Unicode **Encoding** of the file, so characters can be correctly read. If omitted, "ascii" encoding is defaulted. Only little endian character encodings are supported at the moment.

- "utf8"
- "utf16"

- "utf32"
- "ascii" (the default)

Lock_type indicates the way the file is locked for other processes, it is

- **Read** - the file can be opened simultaneously by other processes but not for writing
- **Write** - the file can be opened simultaneously by other processes, but not for writing
- **Read Write** - the file cannot be opened simultaneously by other processes (the default)

filenum Is a valid FreeBASIC file number (in the range 1..255) not being used for any other file presently open. The file number identifies the file for the purpose of file operations. A free file number can be found using the [FreeFile](#) function.

Example

```

Dim ff As UByte
Dim randomvar As Integer
Dim name_str As String
Dim age_ubyte As UByte

ff = FreeFile
Input "What is your name? ", name_str
Input "What is your age? ", age_ubyte
Open "testfile" For Output As #ff
Write #ff, Int(Rnd(0)*42), name_str, age_ubyte
Close #ff
randomvar=0
name_str=""
age_ubyte=0

Open "testfile" For Input As #ff
Input #ff, randomvar, name_str, age_ubyte
Close #ff

```

```
Print "Random Number was: ", randomvar  
Print "Your name is: " + name_str  
Print "Your age is: " + Str(age_ubyte)
```

'File outputted by this sample will look like this
'minus the comment of course:
'23,"Your Name",19

Differences from QB

See also

- **Append**
- **Input (File Mode)**
- **Open**

Specifies that a procedure name can be overloaded

Syntax

```
Declare [Static] Sub procedure_name [cdecl|stdcall|pascal] Overload  
  ([parameter_list]) [Constructor [priority]] [Static] [Export]
```

```
Declare [Static] Function procedure_name [cdecl|stdcall|pascal]  
  "external_name" ([parameter_list]) As return_type [Static] [Export]
```

```
[Public|Private] Sub procedure_name [cdecl|stdcall|pascal] Overload  
  ([parameter_list]) [Constructor [priority]] [Static] [Export]  
  ..procedure body..  
End Sub
```

```
[Public|Private] Function procedure_name [cdecl|stdcall|pascal]  
  "external_name" ([parameter_list]) As return_type [Static] [Export]  
  ..procedure body..  
End Function
```

Description

In procedure declarations, **overload** allows procedure names to be overloaded. Overloaded procedures can then be declared with the same name if their parameter lists are unique if they contain a different number of parameters or different types. Note that this means that two or more procedures can be overloaded if they differ in return type alone.

Once a procedure name has been declared overloaded, further declarations of that name specify **overload**, but it is not required.

overload is not necessary in member procedure declarations, as they are automatically overloaded.

When calling an overloaded procedure, the compiler determines the correct one among a set of compatible candidates, by comparing the argument types with the parameter types specified in the definitions. If no match is found, the compiler generates an error at compile time.

Example

```

Declare Function SUM Overload (A As Integer,B As Integer) As Integer
Function SUM (A As Integer,B As Integer) As Integer
    Function=A+B
End Function
Declare Function SUM Overload (A As Single,B As Single) As Single
Function SUM (A As Single,B As Single) As Single
    Function=A+B
End Function
Dim As Integer A,B
Dim As Single A1,B1
A=2
B=3
A1=2.
b1=3.
Print SUM(A,B)
Print SUM (A1,B1)
Sleep

```

Differences from QB

- New to FreeBASIC

See also

- [Declare](#)
- [Sub, Function](#)

Override



Method attribute; specifies that a method must override a virtual

Syntax

```
Type typename Extends basename
...
Declare Sub|Function|Operator|Property|Destructor ... (
[parameterlist] ) [As datatype] Override
...
End Type
```

Description

In method declarations, **override** can be used to indicate that this method is expected to override a **Virtual** or **Abstract** method from the base class. Then the compiler will show an error if the method does not override anything (only a non-static method can override a virtual or abstract method).

Use of **override** is not mandatory to override a virtual or abstract method, it is highly recommended, as it will help prevent inadvertent errors (name/signature not matching).

override can only be specified on the method declaration in the UDT block, but not on the method body, because it is just a compile-time check in the context of the inheritance hierarchy, and does not affect the method in any way.

override is only recognized as a keyword at the end of member procedure declarations. It can still be used as identifier elsewhere.

Example

```
Type A Extends Object
  Declare Virtual Sub f1( )
  Declare Virtual Function f2( ) As Integer
End Type
```

```
Type B Extends A
    Declare Sub f1( ) Override
    Declare Function f2( ) As Integer Override
End Type

Sub A.f1( )
End Sub

Function A.f2( ) As Integer
    Function = 0
End Function

Sub B.f1( )
End Sub

Function B.f2( ) As Integer
    Function = 0
End Function
```

Differences from QB

- New to FreeBASIC

See also

- **Virtual, Abstract**

Fills an area delimited by a border of a specified color

Syntax

```
Paint [target,] [STEP] (x, y)[, [paint][, [border_color]]]
```

Parameters

target

specifies buffer to draw on.

STEP

indicates that coordinates are relative

(*x*, *y*)

coordinates of the pixel on which to start the flood fill (paint)

paint

the color attribute or fill pattern

a numeric value indicates a color, while a string indicates a fill pattern

border_color

boundary color for the fill

Description

Graphics command to fill an area delimited by a border of specified color

Paint can operate on the current work page as set by the **ScreenSet** S

Filling starts at specified (*x*,*y*) coordinates; if STEP is specified, these coordinates are relative. Coordinates are also affected by custom coordinates system set up by **View** also applies.

If the *paint* argument is a number, it is assumed a color in the same format as the **Color** command. If *paint* is a **string**, the region will be filled with the passed string and the passed string must hold pixels data in a format dependent on the color depth. For 1, 2, 4, 8, 15 and 16 bit color depths, the string must be 64 characters long, and its size should be as follows:

For color depths 1, 2, 4 and 8:

size = 8 * 8 = 64

For color depths 15 and 16:

size = (8 * 8) * 2 = 128

For color depths 24 and 32:
size = (8 * 8) * 4 = 256

If the passed string is smaller, missing pixels will be 0. If the *paint* arg the current foreground color set by *color*. Flood-filling continues until *border_color* is omitted, the current background color is assumed.

Example

```
' draws a white circle painted blue inside
Screen 13
Circle (160, 100), 30, 15
Paint (160, 100), 1, 15
Sleep
```

```
' draws a circle and fills it with a checkered pat
'' choose the bit depth for the Screen
'' try setting this to other values: 8, 16 or 32

Const bit_depth = 8

'' function for returning a pixel color, represent
'' returns a the string in the appropriate format
Function paint_pixel( ByVal c As UInteger, ByVal k

    If bit_depth_ <= 8 Then '' 8-bit:
        Function = Chr( CByte(c) )

    ElseIf bit_depth_ <= 16 Then '' 16-bit:
        Function = MKShort( c Shr 3 And &h1f Or _
                            c Shr 5 And &h7e0 Or _
                            c Shr 8 And &hf800 )

    ElseIf bit_depth_ <= 32 Then '' 32-bit:
        Function = MKL(c)
```

```

        End If

End Function

'' open a graphics window at the chosen bit depth
ScreenRes 320, 200, bit_depth

'' declare variables for holding colors
Dim As UInteger c, c1, c2, cb

'' declare string variable for holding the pattern
Dim As String paint_pattern = ""

'' set colors
If bit_depth <= 8 Then
    c1 = 7 ''pattern color 1
    c2 = 8 ''pattern color 2
    cb = 15 ''border color
Else
    c1 = RGB(192, 192, 192) '' pattern color 1
    c2 = RGB(128, 128, 128) '' pattern color 2
    cb = RGB(255, 255, 255) '' border color
End If

'' make the pattern to be used in Paint
For y As UInteger = 0 To 7
    For x As UInteger = 0 To 7

        '' choose the color of the pixel (c)
        If (x \ 4 + y \ 4) Mod 2 > 0 Then
            c = c1
        Else
            c = c2
        End If

        '' add the pixel to the pattern
        paint_pattern = paint_pattern + paint_pixe

```

```

'' the following line can be used if you w
'' pattern tile in the top left hand corne

' pset (x, y), c

Next x
Next y

'' draw a circle with the border color
Circle (160, 100), 50, cb, , , 1.0

'' paint the circle region with paint_pattern, stc
Paint (160, 100), paint_pattern, cb

'' pause before ending the program
Sleep

```

Differences from QB

- *target* is new to FreeBASIC
- In QB, the fill pattern was always 8-bits wide, and the height was the same as the width; in FreeBASIC, the fill pattern is 8 pixels wide, independent of the color depth,
- The background color parameter supported by QB is not supported in FreeBASIC.

See also

- [Screen](#)

Customizes colors in modes with paletted colors

Syntax

```
Palette [Get] [index, color]  
Palette [Get] [index, r, g, b]  
Palette [Get] Using arrayname(idx)
```

Parameters

Get
indicates getting palette information rather than setting palette information
index
palette index
color
color attribute
r
red color component
g
green color component
b
blue color component
Using
indicates using array of color values
arrayname(idx)
array and index to get/set color attributes

Description

The **Palette** statement is used to retrieve or customize the current palette for graphics modes with a color depth of up to 8bpp; using **Palette** with a mode with a higher color depth will have no effect. Calling **Palette** with no argument restores the default palette for current graphics mode, as seen in the **Screen (Graphics)** statement.

The GfxLib sets a **default palette** when a **screen** mode is initialized.

First form

If you specify index and color, these are dependent on the current mode

Screen mode	index range	color range
-------------	-------------	-------------

1	0-3	0-15
2	0-1	0-15
7,8	0-15	0-15
9	0-15	0-63
11	0-1	see below
12	0-15	see below
13 to 21	0-255	see below

In screen modes 1, 2, 7, 8 and 9 you can assign to each color index or the colors in the available range. In other screen modes, the color must be specified in the form `&h;BBGGRR`, where *BB*, *GG* and *RR* are the blue, green and red components ranging from 0-63; in hexadecimal (0-63 in decimal). If you don't like hexadecimal form, you can use the following formula to convert the integer value to pass to this parameter:

`color = red or (green shl 8) or (blue shl 16)`

Where red, green and blue must range 0-63. Please note that color values accepted by `palette` are **not** the same form as returned by the `palette` macro (the red and blue fields are inverted, and the range is different) for backward compatibility with QB.

Second form

In the second form, you specify the red, green and blue components for each palette entry directly, by calling `palette` with 4 parameters. In this case, *r* and *b* must be in the range 0-255.

Third form

Calling `palette using` allows to set a list of color values all at once; you should pass an array holding enough elements as the color indices available for your current graphics mode color depth (2 for 1bpp, 4 for 2bpp, 16 for 4bpp or 256 for 8bpp). The array elements must be integer color values in the form described above. The colors stored into *arrayname* starting with given *idx* index are then assigned to each palette index, starting with index 0.

Form 1 and 3 are for backward compatibility with QB; form 2 is meant to ease palette handling. Any change to the palette is immediately visible on screen.

If the `get` option is specified, `palette` retrieves instead of setting color for the current palette. The parameters have the same meaning as specified above.

for the form being used, but in this case color, *r*, *g* and *b* must be variables passed by reference that will hold the color RGB values on function exit.

Example

```
' Setting a single color, form 1.  
Screen 15  
Locate 1,1: Color 15  
Print "Press any key to change my color!"  
Sleep  
' Now change color 15 hues to bright red  
Palette 15, &h00003F  
Sleep
```

```
' Getting a single color, form 2.  
Dim As Integer r, g, b  
Screen 13  
Palette Get 32, r, g, b  
Print "Color 32 hues:"  
Print Using "Red:### Green:### Blue:###"; r; g; b  
Sleep
```

```
' Getting whole palette, form 3.  
Dim pal(0 To 255) As Integer  
Screen 13  
Palette Get Using pal  
For i As Integer = 0 To 15  
    Print Using "Color ## = &"; i; Hex(pal(i), 6)  
Next i  
Sleep
```

Differences from QB

- QBasic did not support PALETTE GET to retrieve a palette.
- QBasic did not allow passing individual red/green/blue values.

See also

- **Screen (Graphics)**
- **Color**
- **Using**
- **Internal Pixel Formats**

Specifies a *Pascal*-style calling convention in a procedure declaration

Syntax

```
Sub name pascal [Overload] [Alias "alias"] ( parameters )  
Function name pascal [Overload] [Alias "alias"] ( parameters ) A
```

Description

In procedure declarations, `pascal` specifies that a procedure will use the calling convention, any parameters are to be passed (pushed onto the stack, that is, from left to right. The procedures need not preserve the stack (pop any parameters) before it returns.

`pascal` is not allowed to be used with variadic procedure declarations ("...").

`pascal` is the default calling convention for procedures in Microsoft QuickBasic used in the Windows 3.1 API.

Example

```
Declare Function MyFunc pascal Alias "MyFunc" (MyFunc)
```

Differences from QB

- New to FreeBASIC

See also

- `cdecl`, `stdcall`
- `Declare`
- `Sub`, `Function`

Copies one graphical or text page onto another

Syntax

```
Declare Function PCopy ( ByVal source As Long = -1, ByVal destin
```

Usage

```
PCopy [ source ] [, destination ]
```

Parameters

source
page to copy from
destination
page to copy to

Description

Copies one graphical or text video page to another. Useful for drawing to the active visible page - creating smooth graphics and animation. K

source and *destination* refer to page numbers. The 'source' page is called.

If the *source* argument is omitted, the current working page is assumed visible page is assumed.

`pcopy` is inactive if the *destination* page is locked.

Example

```
'Sets up the screen to be 320x200 in 8-bit color w  
ScreenRes 320, 200, 8, 2
```

```
'Sets the working page to 1 and the displayed page
```

```
ScreenSet 1, 0
```

```
'Draws a circle moving across the top of the screen
```

```
For x As Integer = 50 To 269
```

```
    Cls 'Clears the screen so we can draw a new circle
```

```
    Circle (x, 50), 50, 14 'Draws a yellow circle
```

```
    PCopy 1, 0 'Copies our image from the previous frame
```

```
    Sleep 25 'Waits for 25 milliseconds
```

```
Next x
```

```
'Wait for a keypress before the screen closes
```

```
Sleep
```

```
' Console mode example:
```

```
' Set the working page number to 0, and the visible area
```

```
#if __FB_LANG__ = "QB"
```

```
Screen ,, 0, 1
```

```
#else
```

```
Screen , 0, 1
```

```
#endif
```

```
Dim As Integer i, frames, fps
```

```
Dim As Double t
```

```
t = Timer
```

```
Do
```

```
    ' Fill working page with a certain color and
```

```
    Cls
```

```
    Locate 1, 1
```

```
    Color (i And 15), 0
```

```
    Print String$(80 * 25, Hex$(i, 1));
```

```
    i += 1
```

```
    ' Show frames per second
```

```

Color 15, 0
Locate 1, 1
Print "fps: " & fps,
If Int(t) <> Int(Timer) Then
    t = Timer
    fps = frames
    frames = 0
End If
frames += 1

'' Copy working page to visible page
PCopy

'' Sleep 50ms per frame to free up cpu time
Sleep 50, 1

'' Run loop until the user presses a key
Loop Until Len(Inkey$)

```

Platform Differences

- Maximum number of text pages in Windows is 4.
- Maximum number of text pages in DOS is 8.
- Maximum number of text pages in all other targets is 1.
- Maximum number of graphics pages depends on what was sp called.

Differences from QB

- None

See also

- [ScreenCopy](#)
- [Flip](#)
- [Screen](#)

Gets the value of an arbitrary type at an address in memory

Syntax

```
Declare Function Peek ( ByVal address As Any Ptr ) ByRef As UByte  
Declare Function Peek ( datatype, ByVal address As Any Ptr )  
ByRef As datatype
```

Usage

```
Peek( [ datatype, ] address )
```

Parameters

address

The address in memory to get the value from.

datatype

The type of value to get. If omitted, it defaults to the type of the pointer passed; or to **UByte**, if the address is an **Integer** or an **Any Ptr**.

Description

This procedure returns a reference to the value in memory given by a memory address, and is equivalent to

```
*cast(ubyte ptr, address)
```

or

```
*cast(datatype ptr, address)
```

Example

```
Dim i As Integer, p As Integer Ptr  
p = @i
```

```
Poke Integer, p, 420  
Print Peek(Integer, p)
```

will produce the output:

420

Differences from QB

- **Peek** did not support the *datatype* parameter in QB, and could only return individual bytes.
- **Peek** returns a reference in FB, so can be used to set the memory contents of the address, like with **Operator * (Value Of)**.
- DEF SEG isn't needed anymore because the address space is 32-bit flat in FreeBASIC.

See also

- **Poke**
- **Operator * (Value Of)**

Maps coordinates between view and physical mapping.

Syntax

Declare Function **PMap** (**ByVal** *coord* **As Single**, **ByVal** *func* **As Lon**

Usage

result = **PMap**(*coord*, *func*)

Parameters

coord

An expression indicating the coordinate to be mapped.

func

The mapping function number to be applied to given coordinate.

Return Value

The mapped coordinate value.

Description

This function converts a coordinate between view (as defined by the **View** statement) and physical (as set by the **View (Graphics)** statement) mapping. Depending on the value of *func*, *expr* is used to compute a different mapping returned by **PMap**:

func value:	return value:
0	Treats <i>expr</i> as x view coordinate and returns corresponding x physical coordinate
1	Treats <i>expr</i> as y view coordinate and returns corresponding y physical coordinate
2	Treats <i>expr</i> as x physical coordinate and returns corresponding x view coordinate
3	Treats <i>expr</i> as y physical coordinate and returns corresponding y view coordinate

Example

```
Screen 12
Window Screen (0, 0)-(100, 100)
Print "Logical x=50, Physical x="; PMap(50, 0)
Print "Logical y=50, Physical y="; PMap(50, 1)
Print "Physical x=160, Logical x="; PMap(160, 2)
Print "Physical y=60, Logical y="; PMap(60, 3)
Sleep
```

Differences from QB

- None

See also

- Window
- View (Graphics)

Point



Returns the color attribute of a specified pixel coordinate

Syntax

```
result = Point( coord_x, coord_y [,buffer] )  
or  
result = Point( function_index )
```

Usage

coord_x

x coordinate of the pixel

coord_y

y coordinate of the pixel

buffer

the image buffer to read from

function_index

the type of screen coordinate to return: one of the values 0, 1, 2, 3

Return Value

If the *x*, *y* coordinates of a pixel are provided **Point** returns the color at those coordinates, as an 8-bit palette index in 8 bpp indexed modes, a 24-bit value in 24-bit modes (upper 8 bits of the integer unused, limited precision of R,G,B) or a 32-bit value in 32 bpp modes (upper 8 bits unused or holding Alpha). Note that the return value is a 32-bit value (5 bits R + 6 bits G + 5 bits B).

If the argument is a function index, **Point** returns one of the graphics coordinates from the last graphics command.

Argument	Value Returned
0	The current physical x coordinate.
1	The current physical y coordinate.
2	The current view x coordinate. This returns the same value as the POINT(0) function has not been used.
3	The current view y coordinate. This returns the same value as the POINT(1) function has not been used.

Description

GfxLib Function with two different uses.

If supplied with two coordinates it reads the color of the pixel at the coordinates on the screen, or of the *buffer*, if supplied.

The value returned is a color index in a 256 or less color **Screen**, and an **ImageInfo** modes. If the coordinates are off-screen or off-buffer, -1 is returned

If supplied with a single value it returns the one of the coordinates of the last graphics command executed. If the last command was executed on the buffer, the coordinates returned will be coordinates in the buffer. Arguments out of the range

The function **Point** does not work in text modes.

Speed note: while **Point** provides valid results, it is quite slow to call due to the overhead of additional calculations and checks. Much better performance is achieved using direct memory access using the results obtained from **ImageInfo**

Example

```
' Set an appropriate screen mode - 320 x 240 x 8bpp
ScreenRes 320, 240, 8

' Draw a line using color 12 (light red)
Line (20,20)-(100,100), 12

' Print the color of a point on the line
Print Point(20,20)

' Sleep before the program closes
Sleep
```

Output:

Differences from QB

- *buffer* is new to FreeBASIC
- In 16 bpp and 32 bpp modes, a 32-bit value is returned instead

See also

- **PSet** - write pixels
- **PMap**
- **Color**
- **View (Graphics)**
- **Window**
- **Internal pixel formats**

PointCoord



Queries `Draw`'s pen position in graphics mode

Syntax

```
Declare Function PointCoord( ByVal func As Long ) As Single  
result = PointCoord( func )
```

Description

The `PointCoord` function can be used to query x and y position of the pen in graphics mode. The `result` value depends on the passed `func` value:

func value:	return value:
0	x physical coordinate, same as <code>PMap(PointCoord(2), 0)</code>
1	y physical coordinate, same as <code>PMap(PointCoord(3), 1)</code>
2	x view coordinate
3	y view coordinate

Example

Screen 12

```
Print "--- Default window coordinate mapping ---"  
Print "DRAW pen position, at the default (0,0):"  
Print "Physical:", PointCoord( 0 ), PointCoord( 1 )  
Print "View:", PointCoord( 2 ), PointCoord( 3 )  
  
Draw "BM 50,50"  
Print "DRAW pen position, after being moved to (50,50):"  
Print "Physical:", PointCoord( 0 ), PointCoord( 1 )  
Print "View:", PointCoord( 2 ), PointCoord( 3 )  
  
Print "--- Changing window coordinate mapping ---"  
Window Screen (-100, -100) - (100, 100)
```

```
Draw "BM 0,0"  
Print "DRAW pen position, after being moved to (0,  
Print "Physical:", PointCoord( 0 ), PointCoord( 1  
Print "View:", PointCoord( 2 ), PointCoord( 3 )  
  
Draw "BM 50,50"  
Print "DRAW pen position, after being moved to (50,  
Print "Physical:", PointCoord( 0 ), PointCoord( 1  
Print "View:", PointCoord( 2 ), PointCoord( 3 )  
  
Sleep
```

Differences from QB

- New to FreeBASIC

See also

- PMap
- Window

Pointer



A variable declaration type modifier

Syntax

```
Dim symbolName As DataType {Pointer | Ptr}
```

Description

Declares a pointer variable. The same as **Ptr**.

Example

```
Dim p As ZString Pointer
Dim text As String
text = "Hello World!"
p = StrPtr(text) + 6
Print text
Print *p

'' Output:
'' Hello World!
'' World!
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__Pointer`.

Differences from QB

- New to FreeBASIC

See also

- Ptr

Poke



Assigns a value to a location in memory.

Syntax

```
Declare Sub Poke ( ByVal address As Any Ptr, ByVal value As UByte )  
Declare Sub Poke ( datatype, ByVal address As Any Ptr, ByVal value As datatype )
```

Usage

```
Poke [ datatype, ] address, value
```

Parameters

datatype

The type of data at the specified address.

address

The location in memory to assign to.

value

The value to assign.

Description

Poke assigns a value to a location in memory. It is equivalent to

```
*cast(ubyte ptr, address) = value
```

or

```
*cast(datatype ptr, address) = value
```

When *datatype* is a user-defined type, **Poke** assigns *value* using the type's **Operator Let**.

Example

```
Dim i As Integer, p As Integer Ptr  
p = @i  
  
Poke Integer, p, 420
```

```
Print Peek(Integer, p)
```

Will produce the output:

```
420
```

Differences from QB

- Only the byte form were supported in QB.
- DEF SEG isn't needed anymore because the address space is 32-bit flat in FreeBASIC.

See also

- [Peek](#)

Returns the horizontal (left to right) position of the text cursor

Syntax

```
Declare Function Pos ( ) As Long  
Declare Function Pos ( ByVal dummy As Long ) As Long
```

Usage

```
result = Pos[ ( dummy ) ]
```

Parameters

dummy

An unused parameter retained for backward compatibility with QBASIC.

Return Value

Returns the horizontal position of the text cursor.

Description

Returns the horizontal (left to right) position of the text cursor. The leftmost column is number 1.

Example

```
Dim As Integer p  
  
' print starting column position  
p = Pos()  
Print "position: "; p  
  
' print a string, without a new-line  
Print "ABCDEF";  
  
' print new column position:
```

```
p = Pos()
Print: Print "position: "; p
Print

''position changes after each Print:
Print "Column numbers: "
Print Pos(), Pos(), Pos(), Pos(), Pos()
```

Differences from QB

- The *dummy* parameter was not optional in QBASIC.

See also

- **CsrLin**
- **Tab**
- **Locate**

Preserve



Used with `ReDim` to preserve contents will resizing an array

Syntax

```
ReDim Preserve array(...) [As datatype]
```

Description

Used with `ReDim` so that when an array is resized, data is not reset but is preserved. This means when the array is enlarged that only new data is reset, while the old data remains the same.

NOTE: `ReDim Preserve` may not work as expected in all cases: `Preserve`'s current behavior is to keep the original data contiguous in memory, and only expand or truncate the size of the memory. Its behavior is only well-defined when the upper bound is changed. If the lower bound is changed, the current result is that the data is in effect shifted to start at the new lower bound. If there are multiple dimensions, only the upper bound of the first dimension may be changed safely. If lower-order dimensions are resized at all, the effects can be hard to predict.

Example

```
ReDim array(1 To 3) As Integer
Dim i As Integer

array(1) = 10
array(2) = 5
array(3) = 8

ReDim Preserve array(1 To 10)

For i = 1 To 10
    Print "array("; i; ") = "; array(i)
Next
```

Differences from QB

- Preserve wasn't supported until PDS 7.1

See also

- [Dim](#)
- [LBound](#)
- [ReDim](#)
- [UBound](#)

PRreset



Plots a single pixel

Syntax

```
PRreset [target ,] [STEP] (x, y) [,color]
```

Parameters

target
specifies buffer to draw on.
STEP
indicates that coordinates are relative
(*x*, *y*)
coordinates of the pixel.
color
the color attribute.

Description

target specifies buffer to draw on. *target* may be an image created with `Image`. If omitted, *target* defaults to the screen's current work page.

(*x*, *y*) are the coordinates of the pixel. STEP if present, indicates that graphics cursor position. If omitted, (*x*, *y*) are relative to the upper left corner. Coordinates are affected by the last call to the `View (Graphics)` and `Window (Graphics)` clipping region as set by the `View (Graphics)` statement.

color specifies the color attribute. If omitted, *color* defaults to the current graphics mode specific, see `Color` and `Screen (Graphics)` for details.

Example

```
Screen 13  
  
'Set background color to 15  
Color , 15
```

```
'Draw a pixel with the background color at 10, 10  
PReset (10,10)
```

```
'Draw a pixel with the background color at Last x  
PReset Step (10,10)  
Sleep
```

Differences from QB

- *target* is new to FreeBASIC

See also

- **PSet**

Writes text to the screen

Syntax

```
(Print | ?) [ expressionlist ] [ , | ; ]
```

Parameters

expressionlist
list of items to print

Description

Print outputs a list of values to the screen. Numeric values are converted to string representation, with left padding for the sign. Objects of user-defined type must overload operator `Cast () As String`.

Consecutive values in the expression list are separated either by a comma (,) or a semicolon (;). A comma indicates printing should take place at the next line boundary, while a semicolon indicates values are printed with no space between them. This has a similar effect to concatenating expressions using `+`.

Print also supports the special expressions, **Spc()** and **Tab()**. These expressions are used to space out expressions, or to align the printing to a specific column.

A new-line character is printed after the values in the expression list unless the expression list is followed by a comma or semicolon. A **Print** without any expressions or separators following it will just print a new-line.

NOTE: **Print** resets the **Err** value after each expression is printed.

NOTE: In graphics mode, **Draw String** provides a flexible alternative to **Print**. It prints a string to the screen with pixel positioning, transparent background, and can use a user-supplied font.

Example

```
' ' print "Hello World!", and a new-line
Print "Hello World!"

' ' print several strings on one line, then print a
line
Print "Hello";
Print "World"; "!";
Print

' ' column separator
Print "Hello!", "World!"
```

```
' ' printing variables/expressions
Dim As Double pi = Atn(1) * 4
Dim As String s = "FreeBASIC"

Print "3 * 4 ="; 3 * 4

Print "Pi is approximately"; pi
Print s; " is great!"
```

Dialect Differences

- In the *-lang qb* dialect, an extra space is printed after numbers

Differences from QB

- None, when using QBASIC's variable types in *-lang qb*.
- Unsigned numbers are printed without a space before them.
- QB did not support casting for UDTs, so didn't allow them to be

See also

- `Spc`

- **Tab**
- **(Print | ?) #**
- **(Print | ?) Using**
- **Write**
- **Draw String**
- **Input**

(Print | ?)



Writes a list of values to a file or device

Syntax

```
(Print | ?) # filenum, [ expressionlist ] [ , | ; ]
```

Parameters

filenum

The file number of a file or device opened for **Output** or **Append**.

expressionlist

List of values to write.

Description

Print # outputs a list of values to a text file or device. Numeric values are converted to their string representation, with left padding for the sign. Objects of user-defined types must overload operator `Cast () As String`.

Consecutive values in the expression list are separated either by a comma (,) or semicolon (;). A comma indicates printing should take place at the next 14 column boundary, while a semicolon indicates values are printed with no space between them.

A new-line character is printed after the values in the expression list unless the expression list is followed by a comma or semicolon.

Note that the comma (,) immediately following the file number is still necessary, even the expression list is empty. In this case a new-line is printed, just as with a normal expression list that doesn't have a comma or semicolon at the end.

Example

```
Open "bleh.dat" For Output As #1
```

```
Print #1, "abc def"  
Print #1, 1234, 5678.901, "xyz zzz"  
  
Close #1
```

Dialect Differences

- In the *-lang qb* dialect, an extra space is printed after numbers

Differences from QB

- None, when using QBASIC's variable types in *-lang qb*.
- Unsigned numbers are printed without a space before them.
- QB did not support casting for UDTs, so didn't allow them to be printed.

See also

- (Print | ?) Using
- (Print | ?)
- Write #
- Open

(Print | ?) Using



Outputs formatted text to the screen or output device

Syntax

```
(Print | ?) [# filename ,] [ printexpressionlist {,|;} ] Using fo  
; ] ]
```

Parameters

filename

The file number of a file or device opened for **Output** or **Append**. (Altern where appropriate, instead of **Print #**)

printexpressionlist

Optional preceding list of items to print, separated by commas (,) or s more details).

formatstring

Format string to use.

expressionlist

List of items to format, separated by semi-colons (;).

Description

Print to screen various expressions using a format determined by the Internally, **Print Using** uses a buffer size of 2048 bytes: while it is high be filled, it should be noted that output would be truncated should this

If no expression list is given, the format string will be printed up to the semi-colon after *formatstring* is still necessary, even if no expression

The format string dictates how the expressions are to be formatted w/ indicated by the use of special marker characters. There are markers numeric output:

String formatting

Marker	Formatting
!	prints the first character of a string

\\	prints as many characters of a string as occupied between the pair \\
&	prints the entire string

Numeric formatting

Marker	Formatting
#	placeholder for either an integer digit, or a decimal digit if a decimal point precedes it
,	placed after integer digit indicates groups of 3 digits should be separated by comma
.	placed near # indicates place for the decimal point
^^^	uses exponential notation (E+/-###) when placed after the digit characters
+	placed before/after the format string, controls whether the sign of a number is prepended. '+' sign to be printed for positive numbers
-	placed after the format string, causes the sign of the number to be appended rather than prepended. '-' sign for positive/negative numbers
\$\$	placed at the start of integer digits, causes a dollar sign to be prepended to the number
**	placed at the start of integer digits, causes any padding on the left to be changed from spaces to zeros
**\$	placed at the start of integer digits, pads on the left with asterisks, and prepends a dollar sign
&	prints a number intelligently, using the exact number of digits required (new to version 3.0)

All of the special marker characters can be escaped by preceding them with a backslash "\", allowing them to be printed directly. For example, "\\!" is printed as \!

If a numerical value cannot fit in the number of digits indicated by the format string, the value is printed in scientific notation, and the format string is adapted to fit the number, possibly switching to scientific notation, and the percent "%" character. E.g., the number 1234 with a *formatstring* of "%1234.00".

All other characters within the format string are printed as they appear.

A new-line character is printed after the values in the expression list unless followed by a semicolon (;).

Example

```
Print Using "The value is #.## seconds"; 1.019
Print Using "The ASCII code for the pound sign (_#
Print Using "The last day in the year is & \ \"; 3
```

will produce the output:

```
The value is 1.02 seconds
The ASCII code for the pound sign (#) is 35
The last day in the year is 31 Dec
```

Differences from QB

- QB didn't allow "&" to be used for printing numbers.

See also

- [\(Print | ?\)](#)
- [\(Print | ?\) #](#)
- [Format](#)
- [Using](#)
- [Palette Using](#)

Specifies a procedure having internal linkage

Syntax

```
Private Sub procedure_name [cdecl|stdcall|pascal] [Overload]  
  [Alias "external_name"] [([parameter_list])] [Constructor]  
  [priority]] [Static] [Export]  
  ..procedure body..  
End Sub
```

```
Private Function procedure_name [cdecl|stdcall|pascal] [Overload]  
  [Alias "external_name"] [([parameter_list])] As return_type  
  [Static] [Export]  
  ..procedure body..  
End Function
```

Description

In procedure definitions, **Private** specifies that a procedure has internal linkage, meaning its name is not visible to external modules.

The **option Private** statement allows procedures to be defined with internal linkage by default.

Example

```
'e.g.  
  
Private Sub i_am_private  
End Sub  
  
Sub i_am_public  
End Sub
```

Differences from QB

- New to FreeBASIC

See also

- **Private:** (Access Control)
- **Public**
- **Option Private**
- **Sub**
- **Function**

Private: (Access Control)



Specifies private member access control in a **Type** or **Class**

Syntax

```
Type typename  
Private:  
  member declarations  
End Type
```

Parameters

```
typename  
name of the Type or Class  
member declarations  
declarations for fields, functions, or enumerations
```

Description

Private: indicates that *member declarations* following it have private access function for the **Type** or **Class**.

member declarations following **Private:** are private until a different access

Members in a **Type** declaration are **Public:** by default if no member access

Example

```
Type testing  
  number As Integer  
  Private:  
    nome As String  
  Declare Sub setName( ByRef newnome As String )  
End Type  
  
Sub testing.setName( ByRef newnome As String )  
  ' This is OK. We're inside a member function for testing  
  this.nome = newnome  
End Sub
```

```
Dim As testing myVariable

'' This is OK, number is public
myVariable.number = 69

'' this would generate a compile error
'' - nome is private and we're trying to access it
'' myVariable.nome = "FreeBASIC"
```

Dialect Differences

- Available only in the *-lang fb* dialect.

Differences from QB

- New to FreeBASIC

See also

- **Private**
- **Public:** (Access Control)
- **Protected:** (Access Control)
- **Type**

Operator ProcPtr (Procedure Pointer)



Returns the address of a procedure

Syntax

```
Declare Operator ProcPtr ( ByRef lhs As T ) As T Ptr
```

Usage

```
result = ProcPtr ( lhs )
```

Parameters

lhs

A procedure.

T

The type of procedure.

Return Value

Returns the address of the procedure.

Description

This operator returns the address of a **Sub** or **Function** procedure.

Operator @ (Address Of), when used with procedures, has identical b

Example

```
' This example uses ProcPtr to demonstrate function pointer
Declare Function Subtract( x As Integer, y As Integer) As Integer
Declare Function Add( x As Integer, y As Integer) As Integer
Dim myFunction As Function( x As Integer, y As Integer) As Integer

' myFunction will now be assigned to Add
myFunction = ProcPtr( Add )
Print myFunction(2, 3)
```

```
' myFunction will now be assigned to Subtract. No
myFunction = ProcPtr( Subtract )
Print myFunction(2, 3)

Function Add( x As Integer, y As Integer) As Integer
    Return x + y
End Function

Function Subtract( x As Integer, y As Integer) As Integer
    Return x - y
End Function
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- [Sub](#)
- [VarPtr](#)
- [StrPtr](#)
- [Pointers](#)

Declares or defines a property in a type or class

Syntax

```
{ Type | Class } typename  
Declare Property fieldname ( ) As datatype  
Declare Property fieldname ( [ ByRef | ByVal ] new_value As data  
Declare Property fieldname ( [ ByRef | ByVal ] index As datatype  
Declare Property fieldname ( [ ByRef | ByVal ] index As datatype  
datatype )  
End { Type | Class }
```

```
Property typename.fieldname ( ) As datatype [ Export ]  
statements  
End Property
```

```
Property typename.fieldname ( [ ByRef | ByVal ] new_value As dat  
statements  
End Property
```

```
Property typename.fieldname ( [ ByRef | ByVal ] index As datatyp  
statements  
End Property
```

```
Property typename.fieldname ( [ ByRef | ByVal ] index As datatyp  
datatype ) [ Export ]  
statements  
End Property
```

Parameters

typename

name of the **Type** or **Class**

fieldname

name of the property

new_value

the value passed to property to be assigned

index

the property index value

Description

Property fields are used to get and set values of a **Type** or **Class** in the except instead of a simple assignment to a field or a value retrieved fr

typename is the name of the type for which the **Property** method is dec for *typename* follows the same rules as procedures when used in a **Na**

A **Property** may optionally have one index parameter. When indexed, *fieldname*(Index) = Value.

A hidden **This** parameter having the same type as *typename* is passed used to access the fields of the **Type** or **Class**.

Note: A standard **Property** (get & set) does not work with combination byref get-**Property** (as more generally any result byref function) works

Example

```
Type Vector2D
  As Single x, y
  Declare Operator Cast() As String
  Declare Property Length() As Single
  Declare Property Length( ByVal new_length As Sing
End Type

Operator Vector2D.cast () As String
  Return "(" + Str(x) + ", " + Str(y) + ")"
End Operator

Property Vector2D.Length() As Single
  Length = Sqr( x * x + y * y )
End Property

Property Vector2D.Length( ByVal new_length As Sing
  Dim m As Single = Length
  If m <> 0 Then
    ' ' new vector = old / length * new_length
    x *= new_length / m
    y *= new_length / m
```

```

    End If
End Property

Dim a As Vector2D = ( 3, 4 )

Print "a = "; a
Print "a.length = "; a.length
Print

a.length = 10

Print "a = "; a
Print "a.length = "; a.length

```

Output:

```

a = (3, 4)
a.length = 5

a = (6, 8)
a.length = 10

```

Property Indexing:

```

'' True/False
Namespace BOOL
    Const FALSE = 0
    Const TRUE = Not FALSE
End Namespace

Type BitNum
    Num As UInteger

    '' Get/Set Properties each with an Index.
    Declare Property NumBit( ByVal Index As Integer
    Declare Property NumBit( ByVal Index As Integer,
End Type

'' Get a bit by it's index.

```

```

Property BitNum.NumBit( ByVal Index As Integer ) As Boolean
    Return Bit( This.Num, Index )
End Property

'' Set a bit by it's index.
Property BitNum.NumBit( ByVal Index As Integer, ByVal Value As Boolean ) As Boolean

    '' Make sure index is in Integer range.
    If Index >= ( SizeOf(This.Num) * 8 ) Then
        Print "Out of uInteger Range!"
        Exit Property
    Else
        If Index < 0 Then Exit Property
    End If

    If Value = BOOL.FALSE Then
        This.Num = BitReset( This.Num, Index )
    End If

    If Value = BOOL.TRUE Then
        This.Num = BitSet( This.Num, Index )
    End If

End Property

Dim As BitNum Foo

Print "Testing property indexing with data types:"
Print "F00 Number's Value: " & Foo.Num

'' Set the bit in the number as true.
Foo.NumBit(31) = BOOL.TRUE
Print "Set the 31st bit of F00"

'' Print to see if our bit has been changed.
Print "F00 Number's Value: " & Foo.Num
Print "F00 31st Bit Set? " & Foo.NumBit(31)

```

```
Sleep  
Print ""
```

Output:

```
Testing property indexing with data types:  
F00 Number's Value: 0  
Set the 31st bit of F00  
F00 Number's Value: 2147483648  
F00 31st Bit Set? -1
```

See also

- [Class](#)
- [Type](#)

Protected: (Access Control)



Specifies protected member access control in a **Type** or **Class**

Syntax

```
Type typename  
Protected:  
  member declarations  
End Type
```

Parameters

```
  typename  
  name of the Type or Class  
  member declarations  
  declarations for fields, functions, or enumerations
```

Description

Protected: indicates that *member declarations* following it have protected access. Protected members are accessible only from inside a member function for the **Type** or **Class**, and classes which are derive from the **Type** or **Class**.

member declarations following **Protected:** are protected until a different access control specifier is given, like **Private:** or **Public:**.

Members in a **Type** declaration are **Public:** by default if no member access control specifier is given.

NOTE: This keyword is useful only since fbc version 0.24 because inheritance is then supported.

Example

```
'' Example pending classes feature ...
```

Dialect Differences

- Available only in the *-lang fb* dialect.

Differences from QB

- New to FreeBASIC

See also

- **Class**
- **Private:** (Access Control)
- **Public:** (Access Control)
- **Type**

Plots a single pixel

Syntax

```
PSet [target ,] [STEP] (x, y) [,color]
```

Parameters

target

specifies buffer to draw on.

STEP

indicates that coordinates are relative

(*x*, *y*)

coordinates of the pixel.

color

the color attribute.

Description

target specifies buffer to draw on. *target* may be an image created with `ImageInfo`. *target* defaults to the screen's current work page.

(*x*, *y*) are the coordinates of the pixel. STEP if present, indicates that (*x*, *y*) are relative to the cursor position. If omitted, (*x*, *y*) are relative to the upper left-hand corner of the window affected by the last call to the `View (Graphics)` and `Window` statements by the `View (Graphics)` statement.

color specifies the color attribute, as an 8-bit palette index in 8 bpp in modes (upper 8 bits of the integer unused, limited precision of R,G,B) modes (upper 8 bits unused or holding Alpha). Note that it does NOT affect the Alpha channel. If omitted, *color* defaults to the current foreground color.

Speed note: while `PSet` provides valid results, it is quite slow to call repeatedly. Much better performance can be achieved by using `ImageInfo` and `ScreenInfo/ScreenPtr`.

Example

```
' Set an appropriate screen mode - 320 x 240 x 8bp
ScreenRes 320, 240, 8

' Plot a pixel at the coordinates 100, 100, Color
PSet (100, 100), 15
' Confirm the operation.
Locate 1: Print "Pixel plotted at 100, 100"
' Wait for a keypress.
Sleep

' Plot another pixel at the coordinates 150, 150,
PSet (150, 150), 4
' Confirm the operation.
Locate 1: Print "Pixel plotted at 150, 150"
' Wait for a keypress.
Sleep

' Plot a third pixel relative to the second, Color
' This pixel is given the coordinates 60, 60. It w
' at 60, 60 plus the previous coordinates (150, 15
PSet Step (60, 60), 15
' Confirm the operation.
Locate 1: Print "Pixel plotted at 150 + 60, 150 +
' Wait for a keypress
Sleep

' Explicit end of program
End
```

Differences from QB

- *target* is new to FreeBASIC
- In 16 bpp and 32 bpp modes, a 32-bit value is required instead

See also

- [Point - read out pixels](#)
- [PReset](#)
- [View \(Graphics\)](#)
- [Window](#)
- [Internal pixel formats](#)

PSet



Parameter to the **Put** graphics statement which selects **PSet** as the blitting

Syntax

```
Put [ target, ] [ STEP ] ( x,y ), source [ ,( x1,y1 )-( x2,y2 )
```

Parameters

PSet
Required.

Description

The **PSet** method copies the source pixel values onto the destination p

This is the simplest **Put** method. The pixels in the destination buffer are replaced with the pixels in the source buffer. No additional operations are done on the color values that are treated as transparent. It has the same effect as **Put** individually.

Example

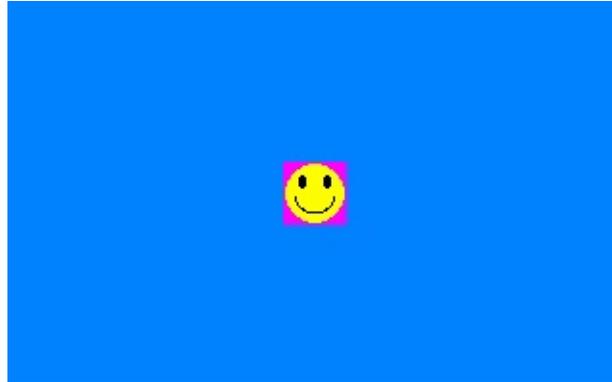
```
' set up a screen: 320 * 200, 16 bits per pixel
ScreenRes 320, 200, 16
Line (0, 0)-(319, 199), RGB(0, 128, 255), bf

' set up an image with the mask color as the background
Dim img As Any Ptr = ImageCreate( 33, 33, RGB(255, 255, 0),
Circle img, (16, 16), 15, RGB(255, 255, 0),
Circle img, (10, 10), 3, RGB( 0, 0, 0),
Circle img, (23, 10), 3, RGB( 0, 0, 0),
Circle img, (16, 18), 10, RGB( 0, 0, 0), 3.14,

Dim As Integer x = 160 - 16, y = 100 - 16

' Put the image with PSET
Put (x, y), img, PSet
```

```
' ' free the image memory  
ImageDestroy img  
  
' ' wait for a keypress  
Sleep
```



Differences from QB

- None

See also

- PSet
- Put (Graphics)

A variable declaration type modifier

Syntax

```
Dim symbolName As DataType {Ptr | Pointer}
```

Description

Declares a pointer variable. The same as **Pointer**.

Operator @ (Address of) operator or **VarPtr** are used to take the address of a variable. **Operator * (Value of)** operator is used to dereference the pointer, that is, to get the value stored in the memory location the pointer is pointing at.

Example

```
' Create the pointer.
Dim p As Integer Ptr

' Create an integer value that we will point to using
Dim num As Integer = 98845

' Point p towards the memory address that variable
p = @num

' Print the value stored in memory pointed to by p
Print "Pointer 'p' ="; *p
Print

' Print the actual location in memory that pointer
Print "Pointer 'p' points to memory location:"
Print p
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- [Pointer](#)
- [Allocate](#)

Specifies a procedure having external linkage.

Syntax

```
Public Sub procedure_name [cdecl|stdcall|pascal] [Overload]  
  [Alias "external_name"] [([parameter_list])] [Constructor  
  [priority]] [Static] [Export]  
  ..procedure body..  
End Sub
```

```
Public Function procedure_name [cdecl|stdcall|pascal] [Overload]  
  [Alias "external_name"] [([parameter_list])] As return_type  
  [Static] [Export]  
  ..procedure body..  
End Function
```

Description

In procedure definitions, **Public** specifies that a procedure has external linkage, meaning its name is visible to external modules. If **Public** or **Private** is not specified, a procedure is defined as if **Public** was specified.

Example

```
Private Sub i_am_private  
End Sub  
  
Public Sub i_am_public  
End Sub
```

Differences from QB

- New to FreeBASIC

See also

- **Public:** (Access Control)
- **Private**
- **Option Private**
- **Sub**
- **Function**

Public: (Access Control)



Specifies public member access control in a **Type** or **Class**

Syntax

```
Type typename  
Public:  
member declarations  
End Type
```

Parameters

```
typename  
name of the Type or Class  
member declarations  
declarations for fields, functions, or enumerations
```

Description

Public: indicates that *member declarations* following it have public access. Public members are accessible with any usage of the **Type** or **Class**.

member declarations following **Public:** are public until a different access control specifier is given, like **Private:** or **Protected:**

Members in a **Type** declaration are **Public:** by default if no member access control specifier is given.

Example

```
Type testing  
  Private:  
    nome As String  
  Public:  
    number As Integer  
  Declare Sub setNome( ByRef newnome As String )  
End Type  
  
Sub testing.setnome( ByRef newnome As String )
```

```
    this.nome = newnome
End Sub

Dim As testing myVariable

'' We can access these members anywhere since
'' they're public
myVariable.number = 69 ''
myVariable.setNome( "FreeBASIC" )
```

Dialect Differences

- Available only in the *-lang fb* dialect.

Differences from QB

- New to FreeBASIC

See also

- **Class**
- **Private:** (Access Control)
- **Protected:** (Access Control)
- **Public**
- **Type**

Put (Graphics)



Copies an image on to another image or screen

Syntax

Put [*target*,] [[STEP](*x*, *y*), *source* [, (*x1*, *y1*)-[STEP](*x2*, *y2*)

Parameters

target

is the address of the buffer where the image is to be drawn. If it's omitted, the graphics cursor position.

[STEP](*x*, *y*)
specify offsets from the upper-left corner of the destination buffer, or source graphics cursor position.

source

is the address of the buffer of the image to be drawn. See below.

(*x1*, *y1*)-[STEP](*x2*, *y2*)

specify a rectangular area in the source buffer to draw. If omitted, the graphics cursor position.

method

specifies the method used to draw the image to the destination buffer.

Background-independent methods

PSet : Source pixel values are copied without modification.

PRESET : Source pixel values are 1's-complement negated before being copied.

Trans : Source pixel values are copied without modification. Does not modify the destination buffer.
Background-dependent methods

And : Destination pixels are bitwise **Anded** with source pixels. See below.

or : Destination pixels are bitwise **ored** with source pixels. See below.

Xor : Destination pixels are bitwise **xored** with source pixels. See below.

Alpha : Source is blended with a transparency factor specified either in the *value* parameter or in the *blender* parameter.

Add: Source is multiplied by a value and added with saturation to the destination buffer.

Custom : Uses a user-defined function to perform blending the source and destination pixels.

value

is a 0. .255 value specifying the transparency value for an ADD or ALPHA method.

blender

specifies the address of a user-defined function to be called in a CUSTOM method.

param

specifies a parameter to pass to the custom blender.

Description

The `Put` statement can be used to draw an image onto another image. A plotted image respects the current clipping region set by last call to the `Put` statement.

Valid Image Buffers

The *source* and *target* image buffers must be valid image buffers. Valid buffers are specified in a `Put` statement using an array name with optional index, `array[index]`.

Drawing methods

Depending on the method used, the existing pixel values in the destination image are preserved. The `TRANS` methods do not use the destination buffer for calculating final pixel values. The other methods will look differently depending on the content of the destination image.

Different pixel formats

The pixel format of an image buffer must be compatible with the current screen mode via the `Screen` statement, the image data may not be valid for the current screen mode. However, you will always be able to draw image buffers onto other image buffers.

The `AND`, `OR` and `XOR` methods produce different results depending on the details of the image buffers.

Mask Color

The `TRANS`, `ALPHA` and `ADD` methods do not draw pixels in the source image (screen) depth: in depths up to 8 bpp (paletted modes) it is equal to color 255). Note that in 32 bpp modes the alpha value of a color does not affect the color. See [Internal pixel formats](#) for details.

Alpha drawing

The `ALPHA` method can be used in two modes. If the *value* parameter is a value of 0 will draw a completely transparent image, whereas a value of 1 will draw the image as is. The `ALPHA` method targets (16 and 32 bpp).

If the *value* parameter is omitted, the `ALPHA` method will take the alpha value of the image (the image can be made more or less transparent than others). This method uses the embedded alpha value in each pixel.

Dealing with the alpha channel

Normally `Put` only allows to draw image buffers onto targets with the same pixel format as the source image.

and the ALPHA method is used, the 8 bpp source image is drawn into the destination image without having to deal with low level access of its pixel data.

Custom Blend Function

The CUSTOM method uses a user-defined function to calculate the final color of the source image, and will receive the source and destination pixel values to draw to the destination buffer. The function has the form:

```
Declare Function identifier ( ByVal source_pixel As UInteger, By
```

identifier is the name of the function. Can be anything.

source_pixel is the current pixel value of the source image.

destination_pixel is the current pixel value of the destination image.

parameter is the parameter that is passed by the PUT command. It should

Example

The following program gives a simple example of how to PUT an image

```
' ' set up the screen and fill the background with
ScreenRes 320, 200, 32
Paint (0, 0), RGB(64, 128, 255)

' ' set up an image and draw something in it
Dim img As Any Ptr = ImageCreate( 32, 32, RGB(255, 255, 0),
Circle img, (16, 16), 15, RGB(255, 255, 0),
Circle img, (10, 10), 3, RGB( 0, 0, 0),
Circle img, (23, 10), 3, RGB( 0, 0, 0),
Circle img, (16, 18), 10, RGB( 0, 0, 0), 3.14,

' ' PUT the image in the center of the screen
Put (160 - 16, 100 - 16), img, Trans

' ' free the image memory
ImageDestroy img

' ' wait for a keypress
Sleep
```

The following example shows how to allocate memory for an image, draw an image, and blend images:

```
Declare Function checked_blend( ByVal src As UIImage, ByVal dest As UIImage, ByVal mode As Integer ) As Boolean

Screen 14, 32

Dim As Any Ptr sprite
Dim As Integer counter = 0

sprite = ImageCreate( 32, 32 )

Line sprite, ( 0, 0 )-( 31, 31 ), RGBA(255, 0, 0, 255)
Line sprite, ( 4, 4 )-( 27, 27 ), RGBA(255, 0, 0, 255)
Line sprite, ( 0, 0 )-( 31, 31 ), RGB(0, 255, 0, 255)
Line sprite, ( 8, 8 )-( 23, 23 ), RGBA(255, 0, 0, 255)
Line sprite, ( 1, 1 )-( 30, 30 ), RGBA(0, 0, 255, 255)
Line sprite, ( 30, 1 )-( 1, 30 ), RGBA(0, 0, 255, 255)

Cls
Dim As Integer i : For i = 0 To 63
    Line( i,0 )-( i,240 ), RGB( i * 4, i * 4, i * 4 )
Next i

'' demonstrate all drawing methods ...
Put( 8,14 ), sprite, PSet
Put Step( 16,20 ), sprite, PRreset
Put Step( -16,20 ), sprite, And
Put Step( 16,20 ), sprite, Or
Put Step( -16,20 ), sprite, Xor
Put Step( 16,20 ), sprite, Trans
Put Step( -16,20 ), sprite, Alpha, 96
Put Step( 16,20 ), sprite, Alpha
Put Step( -16,20 ), sprite, add, 192
```

```

Put Step( 16,20 ), sprite, Custom, @checkedred_b

'' print a description near each demo
Draw String (100, 26), "<- pset"
Draw String Step (0, 20), "<- preset"
Draw String Step (0, 20), "<- and"
Draw String Step (0, 20), "<- or"
Draw String Step (0, 20), "<- xor"
Draw String Step (0, 20), "<- trans"
Draw String Step (0, 20), "<- alpha (uniform)"
Draw String Step (0, 20), "<- alpha (per pixel)"
Draw String Step (0, 20), "<- add"
Draw String Step (0, 20), "<- custom"

ImageDestroy( sprite )
Sleep : End 0

'' custom blender function: chequered put
Function checked_blend( ByVal src As UInteger, E
Dim As Integer Ptr counter
Dim As UInteger pixel

counter = Cast(Integer Ptr, param)
pixel = IIf((( *counter And 4) Shr 2) Xor (( *cou
*counter += 1
Return pixel
End Function

```

Differences from QB

- *target* is new to FreeBASIC
- The TRANS, ALPHA, ADD and CUSTOM methods are new to FreeBAS
- FB uses a different image format internally, which is unsupport

- QB throws a run-time error instead of clipping out-of-bounds in
- In QB, only arrays can be specified as source images

See also

- **Put (File I/O)**
- **Get (Graphics)**
- **ImageCreate**
- **Alpha**
- **Internal pixel formats**

Put (File I/O)



Writes data from a buffer to a file

Syntax

```
Put #filenum As Long, [position As LongInt], data As Any [, amount]
Put #filenum As Long, [position As LongInt], data As String
Put #filenum As Long, [position As LongInt], data() As Any
```

Usage

```
Put #filenum, position, data [, amount]
varres = Put (#filenum, position, data [, amount])
```

Parameters

filenum

The value passed to **open** when the file was opened.

position

Is the position where **Put** must start in the file. If the file was opened **File** given in bytes. If omitted, writing starts at the present file pointer position or byte of a file is at position 1.

If *position* is omitted or zero (0), file writing will start from the current file pointer position.

data

Is the buffer where data is written from. It can be a numeric variable, a string, or an array. If *amount* is omitted, the operation will try to transfer to disk the complete variable, unless *amount* is specified. When putting arrays, *data* should be followed by an empty pair of brackets. *amount* is not allowed.

When putting **strings**, the number of bytes written is the same as the length of the string. *amount* is not allowed.

Note: If you want to write values from a buffer, you should NOT pass a variable name as *data*. (This can be done by dereferencing the variable name.) If you pass a pointer directly, then **Put** will put the memory from the pointer variable into the file.

amount

Makes **Put** write to file *amount* consecutive variables to the file - i.e. it writes *amount* consecutive variables starting at *data*'s location in memory, into the file. If *amount* is omitted it writes a single variable.

Return Value

0 on success; nonzero on error. "disk full" is considered as an error, and if any of data written before is not available, and wouldn't be really useful and

Description

Writes binary data from a buffer variable to a file opened in **Binary** or **Append** mode. **Put** can be used as a function, and will return 0 on success or an error code on failure.

For files opened in **Random** mode, the size in bytes of the data to write is specified by the **length** argument.

Example

```
' Create variables for the file number, and the number of bytes to write
Dim As Integer f
Dim As Long value

' Find the first free file number
f = FreeFile()

' Open the file "file.ext" for binary usage, using Append mode
Open "file.ext" For Binary As #f

value= 10

' Write the bytes of the integer 'value' into the file #f
' starting at the beginning of the file (position 1)
Put #f, 1, value

' Close the file
Close #f
```

```
' Create an integer array
Dim buffer(1 To 10) As Integer
For i As Integer = 1 To 10
    buffer(i) = i
```

Next

```
' Find the first free file file number
Dim f As Integer
f = FreeFile()

' Open the file "file.ext" for binary usage, using
Open "file.ext" For Binary As #f
' Write the array into the file, using file number
' starting at the beginning of the file (position
Put #f, 1, buffer()

' Close the file
Close #f
```

Example

```
Dim As Byte Ptr lpBuffer
Dim As Integer hFile, Counter, Size

Size = 256

lpBuffer = Allocate(Size)
For Counter = 0 To Size-1
    lpBuffer[Counter] = (Counter And &HFF)
Next

' Get free file file number
hFile = FreeFile()

' Open the file "test.bin" in binary writing mode
Open "test.bin" For Binary Access Write As #hFile

' Write 256 bytes from the memory pointed to by
Put #hFile, , lpBuffer[0], Size
```

```
' Close the file
Close #hFile

' Free the allocated memory
Deallocate lpBuffer
```

Differences from QB

- **Put** can write full arrays as in VB or, alternatively, write a multiple location.
- **Put** can be used as a function in FB, to find the success/error code handling procedures.

See also

- **Put (Graphics)** different usage of same keyword
- **Get (File I/O)**
- **Open**
- **Close**
- **Random**
- **Binary**
- **FreeFile**

Specifies file or device to be opened for binary mode

Syntax

`Open filename for Random [Access access_type] [Lock lock_type] a`

Parameters

filename

file name to open

access_type

indicates whether the file may be read from, written to or both

lock_type

locking to be used while the file is open

filenum

unused file number to associate with the open file

record_length

the size of the record used for the file

Description

Opens a file or device for reading and/or writing binary data in the given *record_length*.

If the file does not exist, a new file will be created, otherwise any data already in the file will be lost. The file pointer is initialized by **open** at the start of the file, at record number 1. The file pointer moves in steps of *record_length* bytes.

This file mode uses an user-defined **Type** buffer variable to read/write data. The **Type** variable must be declared and used to include several fields.

The data is saved in binary mode, in the same internal format FreeBASIC uses.

filename must be a string expression resulting in a legal file name in the current directory, unless a path is given.

Access_type - By default **Random** mode allows to both read and write the file. *Access_type* must be one of:

- **Read** - the file is opened for input only
- **Write** - the file is opened for output only
- **Read Write** - the file is opened for input and output (the default)

Lock_type indicates the way the file is locked for other processes (use

- **Shared** - The file can be freely accessed by other processes
- **Lock Read** - The file can't be opened simultaneously for reading
- **Lock Write** - The file can't be opened simultaneously for writing
- **Lock Read Write** - The file cannot be opened simultaneously for reading or writing

If no lock type is stated, the file will be **shared** for other threads of the program and other programs.

Lock and **Unlock** can be used to restrict temporarily access to parts of a file.

filenum is a valid FreeBASIC file number (in the range 1..255) not being used by the system. This number identifies the file for the rest of file operations. A free file number is returned by the **FreeFile** function.

record_length is the amount of bytes the file pointer will move for each record. It is the size of the buffer variable used when **Getting** and **Putting** data. If omitted, it is the size of the buffer variable.

Example

```
' ' This example generates a test file and then lets other programs
' ' that are read live from the file.

Type Entry
    slen As Byte
    sdata As String * 10
End Type

Dim u As Entry
Dim s As String

Open "testfile" For Random As #1 Len = SizeOf(Entry)

' ' Write out 9 records with predefined data
For i As Integer = 1 To 9
    Read s
    u = Type( Len(s), s )
    Put #1, i, u
Next
```

```

Data "., -?!'@:", "abc",      "def"
Data "ghi",      "jkl",      "mno"
Data "pqrs",     "tuv",      "wxyz"

'' Let the user view records by specifying their i
Do
    Dim i As Integer
    Input "Record number: ", i
    If i < 1 Or i > 9 Then Exit Do

    Get #1, i, u
    Print i & ": " & Left( u.sdata, u.slen )
    Print
Loop

Close #1

```

```

Type ScoreEntry Field = 1
    As String * 20 Name
    As Single score
End Type

Dim As ScoreEntry entry

'' Generate a fake boring highscore file
Open "scores.dat" For Random Access Write As #1 Len
For i As Integer = 1 To 10
    entry.name = "Player " & i
    entry.score = i
    Put #1, i, entry
Next
Close #1

'' Read out and display the entries
Open "scores.dat" For Random Access Read As #1 Len

```

```
For i As Integer = 1 To 10
    Get #1, i, entry
    Print i & ":", entry.name, Str(entry.score), e
Next
Close #1
```

Differences from QB

- Care must be taken with dynamic or fixed length strings inside warning at KeyPgType.
- The keyword **Field** can only be used with **Type** to specify the p

See also

- **Open**
- **Binary**
- **Get #**
- **Put #**

Randomize



Seeds the random number generator

Syntax

```
Declare Sub Randomize ( ByVal seed As Double = -1.0, ByVal  
algorithm As Long = 0 )
```

Usage

```
Randomize [ seed ][, algorithm ]
```

Parameters

seed

A **Double** seed value for the random number generator. If omitted, a value based on **Timer** will be used instead.

algorithm

An integer value to select the algorithm. If omitted, the default algorithm for the current **language dialect** is used.

Description

Sets the random seed that helps **Rnd** generate random numbers, and selects the algorithm to use. Valid values for *algorithm* are:

0 - Default for current **language dialect**. This is algorithm **3** in the **-lang fb** dialect, **4** in the **-lang qb** dialect and **1** in the **-lang fblite** dialect.

1 - Uses the C runtime library's `rand()` function. This will give different results depending on the platform.

2 - Uses a fast implementation. This should be stable across all platforms, and provides 32-bit granularity, reasonable degree of randomness.

3 - Uses the Mersenne Twister. This should be stable across all platforms, provides 32-bit granularity, and gives a high degree of randomness.

4 - Uses a function that is designed to give the same random number sequences as QBASIC. This should be stable across all platforms, and provides 24-bit precision, with a low degree of randomness.

5 - Available on Win32 and Linux, using system features (Win32 Crypt API, Linux /dev/urandom) to provide cryptographically random numbers. If those system APIs are unavailable, algorithm 3 will be used instead.

For any given seed, each algorithm will produce a specific, deterministic sequence of numbers for that seed. If you want each call to `Randomize` to produce a different sequence of numbers, a seed that is not quite predictable should be used - for example, the value returned from `Timer`. Omitting the `seed` parameter will use a value based on this. Note: using the `Timer` value directly as a parameter will produce the same seed if used more than once in the same second. However, it is generally not worth calling `Randomize` twice with unpredictable seeds anyway, because the second sequence will be no more random than the first. In most cases, the Mersenne twister should provide a sufficiently random sequence of numbers, without requiring reseeding between `Rnd` calls.

When you call `Randomize` with the QB compatible algorithm, part of the old seed is retained. This means that if you call `Randomize` several times with the same seed, you will **not** get the same sequence each time. To get a specific sequence in QB compatible mode, set the seed by calling `Rnd` with a negative parameter.

Example

```
' ' Seed the RNG to the method using C's rand()
Randomize , 1

' ' Print a sequence of random numbers
For i As Integer = 1 To 10
    Print Rnd
Next
```

Dialect Differences

The default algorithm used depends on the current dialect in use:

- With the **-lang fb** dialect, a 32 bit Mersenne Twister function with a granularity of 32 bits is used.
- With the **-lang qb** dialect, a function giving the same output as `Rnd` in QB is used. The granularity is 24 bits.
- With the **-lang deprecated** and **-lang fblite** dialects, the function in the C runtime available in the system is used. The function has a granularity of 15 bits in Win32, and 3 bits in Linux and DOS.

Differences from QB

- The *algorithm* parameter is new to FreeBASIC.
- QBASIC only had one algorithm (replicated in FB in algorithm number 4, and set as the default in the **-lang qb** dialect).

See also

- [Rnd](#)
- [Language dialects](#)

Read



Reads values stored with the **Data** statement.

Syntax

```
Read variable_list
```

Description

Reads data stored in the application with the **Data** command.

The elements of the *variable_list* must be of basic types, numeric, strings, arrays and user defined types.

All the **Data** statements in the program behave as a single list, after the **Data** statement is read, the first element of the following **Data** statement is read. The program should not attempt to **Read** after the last **Data** element. This is undefined behavior in some dialects, and the program may crash (Page Fault).

Data constants can only be of simple types (numeric or string). A string variable will be evaluated by the **Val** function.

The "**Restore** *label*" statement makes the first **Data** item after *label* the first to be read, allowing the user to choose specific sections of data to be read.

Example

```
' Create an array of 5 integers and a string to hold them
Dim As Integer h(4)
Dim As String hs
Dim As Integer readindex

' Set up to loop 5 times (for 5 numbers... check to see if they are prime)
For readindex = 0 To 4

    ' Read in an integer.
    Read h(readindex)
```

```

' Display it.
Print "Number" ; readindex ; " = " ; h(readindex

Next readindex

' Spacer.
Print

' Read in a string.
Read hs

' Print it.
Print "String = " + hs

' Await a keypress.
Sleep

' Exit program.
End

' Block of data.
Data 3, 234, 4354, 23433, 87643, "Bye!"

```

Dialect Differences

- None in syntax and usage of **Read**
- See the **Data** page for more information on differences in storin

Differences from QB

- None in syntax and usage of **Read**
- See the **Data** page for more information on differences in storin

See also

- Data
- Restore

Read (File Access)



File access specifier

Syntax

Open *filename* As String For Binary Access **Read** As #*filenum* As Integer

Description

Specifier for the **Access** clause in the **Open** statement. **Read** specifies that the file is accessible for input.

Example

See example at [Access](#)

Differences from QB

- None known.

See also

- [Access](#)
- [Open](#)

Read Write (File Access)



File access specifier

Syntax

Open *filename* As String For Binary Access **Read Write** As #*filenum*
As Integer

Description

Specifier for the **Access** clause in the **Open** statement. **Read Write** specifies that the file is accessible for both input and output.

Example

See example at [Access](#)

Differences from QB

- None known.

See also

- [Access](#)
- [Open](#)

Reallocate



Reallocates storage for an existing reserved block of memory

Syntax

Declare Function Reallocate cdecl (**ByVal** *pointer* **As Any Ptr**, **By**

Usage

```
result = Reallocate( pointer, count )
```

Parameters

pointer

The address of allocated memory to be reallocated.

count

The number of bytes, in total, to be reallocated.

Return Value

The address of the reallocated memory. A null (0) pointer is returned if pointed to by *pointer* remains unchanged.

Description

Attempts to reallocate, or resize, memory previously allocated with **Allocate**. The original memory is preserved, although if *count* is less than the original size of the memory, the added memory range is not initialized to anything.

When using **Reallocate**, the *result* pointer must be saved to prevent it no longer be valid after reallocation. The value of the new pointer should be saved. The original *pointer* remains valid, and the amount of memory allocated to it remains the same.

Reallocated memory must be freed with **Deallocate** when no longer needed.

If *pointer* is null (0), then **Reallocate** behaves identically to **Allocate**. If *count* is 0, **Reallocate** behaves similar to **Deallocate** and a null (0) pointer is returned.

If the memory has previously been deallocated by a call to **Deallocate**, **Reallocate** will return a null (0) pointer.

When manually allocating memory for **String** descriptors (or **udts** that memory block, the new extra memory range must be explicitly cleared (**clear**). Otherwise accessing the string will cause undefined results (trying to deallocate a random pointer).

This function is not part of the FreeBASIC runtime library, it is an alias to be thread safe in all platforms.

NOTE: Reallocating a pointer inside an object function, when that pointer is used and will likely result in horrible crashes.

Example

```
Dim a As Integer Ptr, b As Integer Ptr, i As Integer
a = Allocate( 5 * SizeOf(Integer) ) ' Allocate memory
If a = 0 Then Print "Error Allocating a": End

For i = 0 To 4
    a[i] = (i + 1) * 2 ' Assign integers to the buffer
Next i

b = Reallocate( a, 10 * SizeOf(Integer) ) ' Reallocate memory
If b <> 0 Then
    a = b ' Discard the old pointer and use the new one
    For i = 5 To 9
        a[i] = (i + 1) * 2 ' Assign more integers
    Next i

    For i = 0 To 9 ' Print the integers
        Print i, a[i]
    Next i
    Print
```

```
Else ' ' Reallocate failed, memory unchanged

Print "Error Reallocating a"

For i = 0 To 4 ' Print the integers
  Print i, a[i]
Next i
Print

End If

Deallocate a ' Clean up
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- **Allocate**
- **CAllocate**
- **Deallocate**

Defines or resizes a variable-length array

Syntax

Declaring a Dynamic Array:

```
ReDim [ Shared ] symbolName([subscript [, ...]]) As datatype [,
ReDim [ Shared ] As datatype symbolName([subscript [, ...]]) [,
```

Resizing a Dynamic Array:

```
ReDim [ Preserve ] symbolName([subscript [, ...]]) [, ...]
```

Parameters

Shared

Specifies shared (file-scope) access to the array throughout the module.

Preserve

When used with an existing array, the contents of the array will be preserved.

Preserve will not preserve data at its original index, see below.

symbolName

A new or existing array id.

subscript: [*lowerbound* To] *upperbound*

The lower and upper bound range for a dimension of the array. Lower bound must be specified.

datatype

The type of elements contained in the array.

Description

ReDim can be used to define new variable-length arrays, or resize existing arrays. **ReDim** always produces variable-length arrays, with constant subscripts.

When defining a new variable-length array, its elements are default **Double**, the elements are initialized to zero (0). For user-defined types

NOTES:

- **ReDim Preserve** may not work as expected in all cases: **Preserve**'s current behavior is to keep the original data contiguous in memory.

memory.

Its behavior (with a single dimension) is well-defined only when the upper bound of the current result is that the data is in effect shifted to start at the new lower bound. With multiple dimensions, only the upper bound of only the first dimension is reduced, the existing mappable data may be lost. If lower-order dimensions are reduced, predict.

- **ReDim** cannot be used on fixed-size arrays - i.e. arrays with fixed-size arrays contained in UDTs (user-defined **Types**) as parameters in a function. FreeBASIC cannot prevent you from doing so, and time will be undefined.
- Using **ReDim** within a member procedure with an array that is not yet defined, and will [hopefully] result in horrible crashes.
- For use of **ReDim** (resizing) with a complex expression, (such as a function call), the array expression must be enclosed in parentheses.

Example

```
' ' Define a variable-length array with 5 elements
' '
ReDim array(0 To 4) As Integer

For index As Integer = LBound(array) To UBound(array)
    array(index) = index
Next

' ' Resize a variable-length array with 10 elements
' ' (the lower bound should be kept the same)
ReDim Preserve array(0 To 9) As Integer

Print "index", "value"
For index As Integer = LBound(array) To UBound(array)
    Print index, array(index)
Next
```

This program will produce the following output:

index	value
0	0
1	1
2	2
3	3
4	4
5	0
6	0
7	0
8	0
9	0

```

'' Define a variable-length array
Dim array() As Integer

'' ReDim array to have 3*4 elements
ReDim array(1 To 3, 1 To 4)

Dim As Integer n = 1, i, j

Print "3 * 4:"
Print
For i = LBound(array, 1) To UBound(array, 1)
    For j = LBound(array, 2) To UBound(array, 2)
        array(i, j) = n
        Print Using "## "; array(i, j);
        n += 1
    Next
    Print
Next
Print

'' ReDim Preserve array to have 4*4 elements, pres
'' (only the first upper bound should be changed)
ReDim Preserve array(1 To 4, 1 To 4) As Integer

Print "4 * 4:"
Print

```

```

For i = LBound(array, 1) To UBound(array, 1)
    For j = LBound(array, 2) To UBound(array, 2)
        Print Using "## "; array(i, j);
    Next
    Print
Next
Print

'' ReDim Preserve array to have 2*4 elements, pres
'' (only the first upper bound should be changed)
ReDim Preserve array(1 To 2, 1 To 4) As Integer

Print "2 * 4:"
Print
For i = LBound(array, 1) To UBound(array, 1)
    For j = LBound(array, 2) To UBound(array, 2)
        Print Using "## "; array(i, j);
    Next
    Print
Next
Print

```

This program will produce the following output:

```

3 * 4:

1  2  3  4
5  6  7  8
9 10 11 12

4 * 4:

1  2  3  4
5  6  7  8
9 10 11 12
0  0  0  0

2 * 4:

```

1	2	3	4
5	6	7	8

Differences from QB

- **Preserve** was in Visual Basic, but not in QBASIC.
- Multi-dimensional arrays in FreeBASIC are in row-major order,

See also

- **Common**
- **Dim**
- **Erase**
- **Extern**
- **LBound**
- **Preserve**
- **Shared**
- **Static**
- **UBound**
- **Var**

Indicates comments in the source code.

Syntax

Rem comment

' Comment

/' Multi-line
comment '/

Description

A source code line beginning with **Rem** indicates that the line is a comment and will not be compiled.

The single quote character (') may also be used to indicate a comment may appear after other keywords on a source line.

Multi-line comments are marked with the tokens '/' and '/'. All text between the two markers is considered comment text and is not compiled.

Example

```
/' this is a multi line  
comment as a header of  
this example '/  
  
Rem This Is a Single Line comment  
  
' this is a single line comment  
  
? "Hello" : Rem comment following a statement  
  
Dim a As Integer ' comment following a statement  
  
? "FreeBASIC" : ' also acceptable
```

```
Dim b As /' can comment in here also '/ Integer
#if 0
    This way of commenting Out code was
    required before version 0.16
#endif
```

Differences from QB

- Multiline comments are new to FreeBASIC

See also

- `#if`

Reset



Closes all open files, or resets standard I/O handles.

Syntax

```
Declare Sub Reset ( )  
Declare Sub Reset ( ByVal streamno As Long )
```

Usage

```
Reset  
or  
Reset( streamno )
```

Parameters

streamno

The stream number to reset, 0 for stdin or 1 for stdout.

Description

`Reset`, when called with no arguments, closes all disk files.

`Reset`, when called with the *streamno* argument, will reset the redirected or piped streams associated with stdin (0), or stdout (1).

Runtime errors:

`Reset(streamno)` can set one of the following **runtime errors**:

(1) *Illegal function call*

- *streamno* was neither 0 nor 1

(3) *File I/O error*

- Resetting of stdin or stdout failed

Example

```
Open "test.txt" For Output As #1
```

```
Print #1, "testing 123"  
Reset
```

```
Dim x As String  
  
' Read from STDIN from piped input  
Open Cons For Input As #1  
While EOF(1) = 0  
    Input #1, x  
    Print "*****"; x; "*****"  
Wend  
Close #1  
  
' Reset to read from the keyboard  
Reset(0)  
  
Print "Enter some text:"  
Input x  
  
' Read from STDIN (now from keyboard)  
Open Cons For Input As #1  
While EOF(1) = 0  
    Input #1, x  
    Print "*****"; x; "*****"  
Wend  
Close #1
```

Note: Under Windows, to specify to the program that data entry is completed (transfer EOF), you can press CTRL+Z then press ENTER

Differences from QB

- None for `Reset()`.
- The `Reset(streamno)` usage is new to FreeBASIC.

See also

- **Close**
- **Open**
- **Open Cons**
- **Isredirected**

Restore



Changes the next read location for values stored with the `Data` statement

Syntax

```
Restore label
```

Description

Sets the next-data-to-read pointer to the first element of the first `Data` label must be contained in the same module as the currently-executing normal top to bottom order in which `Data` are `Read`. It allows re-reading sets of data in a single module.

Example

```
' Create an 2 arrays of integers and a 2 strings t
Dim h(4) As Integer
Dim h2(4) As Integer
Dim hs As String
Dim hs2 As String
Dim read_data1 As Integer
Dim read_data2 As Integer

' Set the data read to the label 'dat2:'
Restore dat2

' Set up to loop 5 times (for 5 numbers... check t
For read_data1 = 0 To 4

    ' Read in an integer.
    Read h(read_data1)

    ' Display it.
    Print "Bloc 1, number"; read_data1;" = "; h(read_data1)

Next
```

```

' Spacer.
Print

' Read in a string.
Read hs

' Print it.
Print "Bloc 1 string = " + hs

' Spacers.
Print
Print

' Set the data read to the label 'dat1:'
Restore dat1

' Set up to loop 5 times (for 5 numbers... check t
For read_data2 = 0 To 4

    ' Read in an integer.
    Read h2(read_data2)

    ' Display it.
    Print "Bloc 2, number"; read_data2;" = "; h2(rea

Next

' Spacer.
Print

' Read in a string.
Read hs2

' Print it.
Print "Bloc 2 string = " + hs2

' Await a keypress.

```

Sleep

```
' Exit program.
```

```
End
```

```
' First block of data.
```

```
dat1:
```

```
Data 3, 234, 4354, 23433, 87643, "Bye!"
```

```
' Second block of data.
```

```
dat2:
```

```
Data 546, 7894, 4589, 64657, 34554, "Hi!"
```

Differences from QB

- None

See also

- [Data](#)
- [Read](#)

Error handling statement to resume execution after a jump to an error handler.

Syntax

Resume

Description

Resume is used in the traditional QB error handling mechanism within a line that caused the error. Usually this is used after the error has been corrected to resume operation again with corrected data.

Resume resets the **Err** value to 0

Example

```
' ' Compile with -lang fblite or qb
#lang "fblite"

Dim As Single i, j

On Error Goto ErrorHandler

i = 0
j = 1 / i ' this line causes a divide-by-
zero error on the first try; execution jumps to Er

Print j ' after the value of i is corrected, print

End ' end the program so that execution does not f

ErrorHandler:

i = 2
Resume ' execution jumps back to 'j = 1 / i' line,
```

Dialect Differences

- RESUME is not supported in the *-lang fb* dialect. Statements c

```
If Open( "text" For Input As #1 ) <> 0 Then
  Print "Unable to open file"
End If
```

Differences from QB

- Does not accept line numbers or labels
- Must compile with *-ex* option

See also

- [Err](#)
- [Resume Next](#)
- [Error Handling](#)

Return



Control flow statement to return from a procedure or **GoSub**.

Syntax

```
Return [ expression ]  
or  
Return [ label ]
```

Description

Return is used to return from a procedure or return from a gosub **GoSub**

Because **Return** could mean return-from-gosub or return-from-procedure, it can be used to enable and disable **GoSub** support. When **GoSub** support is not recognized as return-from-procedure. When **GoSub** support is enabled, **Return** is used to return from-gosub.

Return (from procedure) is used inside a procedure to exit the procedure. It cannot specify a return value. In a **Function**, **Return** must specify a value. It is roughly equivalent to the `Function = expression : Exit` Function is used to return from-gosub.

Return (from gosub) is used to return control back to the statement immediately before the call. When used in combination with **GoSub**, no return value can be specified, execution continues at the specified label. If no **GoSub** was made, and execution continues immediately after **Return**.

A **GoSub** should always have a matching **Return** statement. However, if no **GoSub** was made, a run-time error is generated.

Example

```
' ' GOSUB & RETURN example, compile with "-lang qb"  
'$lang: "qb"  
  
Print "Let's Gosub!"
```

```
GoSub MyGosub
Print "Back from Gosub!"
Sleep
End
```

```
MyGosub:
Print "In Gosub!"
Return
```

```
' ' Return from function

Type rational          ' ' simple rational number
    numerator As Integer
    denominator As Integer
End Type

' ' multiplies two rational types
Function rational_multiply( r1 As rational, r2 As rational ) As rational

    Dim r As rational
    ' ' multiply the divisors ...
    r.numerator    = r1.numerator    * r2.numerator
    r.denominator  = r1.denominator  * r2.denominator

    ' ' ... and return the result
    Return r

End Function

Dim As rational r1 = ( 6, 105 )    ' ' define some r
Dim As rational r2 = ( 70, 4 )
Dim As rational r3

r3 = rational_multiply( r1, r2 )    ' ' multiply and

' ' display the expression
```

```
Print r1.numerator & "/" & r1.denominator; " * ";  
Print r2.numerator & "/" & r2.denominator; " = ";  
Print r3.numerator & "/" & r3.denominator
```

Dialect Differences

- In the *-lang fb* dialect **Return** always means return-from-procedure
- In the *-lang qb* dialect, **Return** means return-from-gosub by default **Nogosub**, in which case the compiler will recognize **Return** as return-from-gosub
- In the *-lang fblite* dialect, **Return** means return-from-procedure **Option Gosub**, in which case the compiler will recognize **Return** as return-from-procedure

Differences from QB

- None when using the *-lang qb* dialect.

See also

- **Sub**
- **Function**
- **GoSub**
- **Option Gosub**
- **Option Nogosub**

Computes a valid color value for hi/truecolor modes

Syntax

```
#define RGB(r,g,b) ((CUInt(r) Shl 16) Or (CUInt(g) Shl 8) Or CUI
```

Usage

```
result = RGB(red, green, blue)
```

Parameters

red
red color component value
green
green color component value
blue
blue color component value

Return Value

The combined color.

Description

red, *green* and *blue* are components ranging 0-255.

The **RGB** function can be used to compute a valid color value for use w integer in the format `&h;AARRGGBB`, where *RR*, *GG* and *BB* equal the value *AA* is the implicit alpha value and is automatically set to `&hFF`; (opaque It is possible to retrieve the red, green, blue and alpha values from a c **shr**. The second example below shows how to **#define** and use macro

Note for Windows API programmers: The macro named **RGB** in the \ the FB headers for Windows to avoid collisions.

Example

See `Put (Graphics)` example in addition.

```
ScreenRes 640,480,32 '32 bit color
Line(0,0)-
(319,479), RGB(255,0,0) 'draws a bright red box on
Line(639,0)-
(320,479), RGB(0,0,255) 'draws a bright blue box on

Sleep 'wait before exiting
```

```
' ' setting and retrieving Red, Green, Blue and Alpha

#define RGBA_R( c ) ( CUInt( c ) Shr 16 And 255 )
#define RGBA_G( c ) ( CUInt( c ) Shr  8 And 255 )
#define RGBA_B( c ) ( CUInt( c )           And 255 )
#define RGBA_A( c ) ( CUInt( c ) Shr 24           )

Dim As UInteger r, g, b, a

Dim As UInteger col = RGB(128, 192, 64)

Print Using "Color: _&H\          \"; Hex(col, 8)

r = RGBA_R( col )
g = RGBA_G( col )
b = RGBA_B( col )
a = RGBA_A( col )

Print
Print Using "Red:          _&H\ = ###"; Hex(r, 2);
Print Using "Green:       _&H\ = ###"; Hex(g, 2);
Print Using "Blue:        _&H\ = ###"; Hex(b, 2);
Print Using "Alpha:       _&H\ = ###"; Hex(a, 2);
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- `RGBA`
- `Color`
- `#define`

Computes a valid color value including alpha (transparency) for hi/truecolor modes.

Syntax

```
#define RGBA(r,g,b,a) ((CUInt(r) Shl 16) Or (CUInt(g) Shl 8) Or  
24))
```

Usage

```
result = RGBA(red, green, blue, alpha)
```

Parameters

red
red color component value
green
green color component value
blue
blue color component value
alpha
alpha component value

Return Value

the combined color

Description

red, *green*, *blue* and *alpha* are components ranging 0-255.

The **RGBA** function can be used to compute a valid color value including alpha in hi/truecolor modes. It returns an unsigned integer in the format `&h;A` equal the values passed to this function, in hexadecimal format.

It is possible to retrieve the red, green, blue and alpha values from a color combination of **And** and **Shr**. The second example below shows how to do this.

Example

```

'open a graphics screen (320 * 240, 32-bit)
ScreenRes 320, 240, 32

Dim As Any Ptr img
Dim As Integer x, y

'make an image that varies in transparency and color
img = ImageCreate(64, 64)
For x = 0 To 63
    For y = 0 To 63
        PSet img, (x, y), RGBA(x * 4, 0, y * 4, (x + y))
    Next y
Next x
Circle img, (31, 31), 25, , , RGBA(0, 127, 192, 1)
transparent blue circle
Line img, (26, 20)-(
(38, 44), RGB(255, 255, 255, 0), BF 'transparent

'draw a background (diagonal white lines)
For x = -240 To 319 Step 10
    Line (x, 0)-(319, 240), RGB(255, 255, 255)
Next

Line (10, 10)-(310, 37), RGB(127, 0, 0), BF 'red
Line (10, 146)-
(310, 229), RGB(0, 127, 0), BF 'green box for Putt

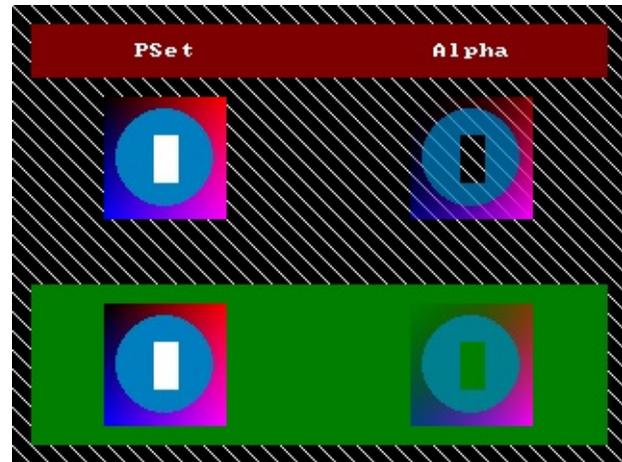
'draw the image and some text with PSET
Draw String(64, 20), "PSet"
Put(48, 48), img, PSet
Put(48, 156), img, PSet

'draw the image and some text with ALPHA
Draw String (220, 20), "Alpha"
Put(208, 48), img, Alpha
Put(208, 156), img, Alpha

```

```
'Free the image memory  
ImageDestroy img
```

```
'Keep the window open until the user presses a key  
Sleep
```



```
' ' setting and retrieving Red, Green, Blue and Alpha  
  
#define RGBA_R( c ) ( CUInt( c ) Shr 16 And 255 )  
#define RGBA_G( c ) ( CUInt( c ) Shr  8 And 255 )  
#define RGBA_B( c ) ( CUInt( c )           And 255 )  
#define RGBA_A( c ) ( CUInt( c ) Shr 24           )  
  
Dim As UInteger r, g, b, a  
  
Dim As UInteger col = RGBA(255, 192, 64, 128)  
  
Print Using "Color: _&H\          \"; Hex(col, 8)  
  
r = RGBA_R( col )  
g = RGBA_G( col )  
b = RGBA_B( col )  
a = RGBA_A( col )
```

```
Print
Print Using "Red:      _&H\\ = ###"; Hex(r, 2);
Print Using "Green:   _&H\\ = ###"; Hex(g, 2);
Print Using "Blue:    _&H\\ = ###"; Hex(b, 2);
Print Using "Alpha:   _&H\\ = ###"; Hex(a, 2);
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- [RGB](#)
- [Color](#)
- [#define](#)

Returns the rightmost substring of a string

Syntax

```
Declare Function Right ( ByRef str As Const String, ByVal n As Integer ) As String  
Declare Function Right ( ByRef str As Const WString, ByVal n As Integer ) As WString
```

Usage

```
result = Right[$]( str, n )
```

Parameters

str
The source string.

n
The substring length, in characters.

Return Value

Returns the rightmost substring from *str*.

Description

Returns the rightmost *n* characters starting from the right (end) of *str*. If *str* is empty, then the null string ("") is returned. If *n* <= 0 then the null string ("") is returned. If *n* > len(*str*) then the entire source string is returned.

Example

```
Dim text As String = "hello world"  
Print Right(text, 5)
```

will produce the output:

```
world
```

An Unicode example:

```
dim text as wstring*20  
text = "Привет, мир!"  
print right(text, 5) 'displays " мир!"
```

Platform Differences

- DOS does not support the wide-character string version of `Right`.

Dialect Differences

- The string type suffix "\$" is obligatory in the *-lang qb* dialect.
- The string type suffix "\$" is optional in the *-lang fblite* and *-lang fb* dialects.

Differences from QB

- QB does not support Unicode.

See also

- `Left`
- `Mid (Function)`

Removes a folder/directory from the file system

Syntax

```
Declare Function Rmdir ( ByRef folder As Const String ) As Long
```

Usage

```
result = Rmdir( folder )
```

Parameters

folder

The folder/directory to be removed.

Return Value

Returns zero (0) on success, and negative one (-1) on failure.

Description

Removes a folder from the file system. The function will fail if the folder

Example

```
Dim pathname As String = "foo\bar\baz"  
Dim result As Integer = Rmdir( pathname )  
  
If 0 <> result Then Print "error: unable to remove"
```

Platform Differences

- Linux requires the *folder* case matches the real name of the file
- Path separators in Linux are forward slashes / . Windows uses backward \ slashes.

Differences from QB

- None

See also

- [Shell](#)
- [ChDir](#)
- [MkDir](#)

Rnd



Returns a random **Double** precision number in the range [0, 1)

Syntax

```
Declare Function Rnd ( ByVal seed As Single = 1.0 ) As Double
```

Usage

```
result = Rnd( seed )
```

Parameters

seed

Optional **single** argument. If *seed* has a value of zero (0.0), the last random number a new random number is returned. With the QB-compatible a generator. The default for no argument is to return a new random number.

Return Value

Returns the random number generated.

Description

Returns a number of type **double** in the range [0, 1) (i.e. $0 \leq \text{Rnd} < 1$)

Rnd can use a variety of different algorithms - see **Randomize** for details

Rnd will return the same sequence of numbers every time a program is run using the generator.

Example

```
' ' Function to a random number in the range [first, last)
Function rnd_range (first As Double, last As Double) As Double
    Function = Rnd * (last - first) + first
End Function
```

```

'' seed the random number generator, so the sequer
Randomize

'' prints a random number in the range [0, 1), or
Print Rnd

'' prints a random number in the range [0, 10), or
Print Rnd * 10

'' prints a random integral number in the range [1
'' with integers, this is equivalent to [1, 10], c
Print Int(Rnd * 10) + 1

'' prints a random integral number in the range [6
'' this is equivalent to [69, 420], or {69 <= n <=
Print Int(rnd_range(69, 421))

```

Dialect Differences

The default algorithm used depends on the current dialect in use:

- With the *-lang fb* dialect, a 32 bit Mersenne Twister function is used.
- With the *-lang qb* dialect, a function giving the same output as the Mersenne Twister function is used.
- With the *-lang deprecated* and *-lang fblite* dialects, the Mersenne Twister function is used. The function available in Win32 has a granularity of 1.

Differences from QB

- None, if compiled in the *-lang qb* dialect. Other dialects can also use the Mersenne Twister function by calling `Randomize` with the appropriate parameter.
- For the non-QB-compatible algorithms, if the optional argument is not passed, the function will return a random number in the range [0, 1). If an argument is passed, the function will return a random number in the range [0, argument), or [argument, 420] if the argument is greater than 420.

See also

- `Randomize`
- `Timer`

- Int

Right justifies a string in a string buffer

Syntax

```
Declare Sub RSet ( ByRef dst As String, ByRef src As Const String )
Declare Sub RSet ( ByVal dst As WString Ptr, ByVal src As Const WString Ptr )
```

Usage

```
RSet dst, src
```

Parameters

dst

A **String** or **WString** buffer to copy the text into.

src

The source **String** or **WString** to be right justified.

Description

RSet right justifies text into the string buffer *dst*, filling the right part of the string with *src* and the left part with spaces. The string buffer size is not modified.

If text is too long for the string buffer size, **RSet** truncates characters from the right.

Example

```
Dim buffer As String
buffer = Space(10)
RSet buffer, "91.5"
Print "-[" & buffer & "]"
```

Differences from QB

- In QBasic the syntax was `RSet dst = src`. That syntax is also supported by FB.

See also

- LSet
- Space
- Put (File I/O)
- MKD
- MKI
- MKL
- MKS

Removes surrounding substrings or characters on the right side of a string

Syntax

```
Declare Function RTrim ( ByRef str As Const String, [ Any ] ByRef  
trimset As Const String = " " ) As String  
Declare Function RTrim ( ByRef str As Const WString, [ Any ]  
ByRef trimset As Const WString = WStr(" ") ) As WString
```

Usage

```
result = RTrim[$]( str [, [ Any ] trimset ] )
```

Parameters

str

The source string.

trimset

The substring to trim.

Return Value

Returns the trimmed string.

Description

This procedure trims surrounding characters from the right (end) of a source string. Substrings matching *trimset* will be trimmed if specified otherwise spaces (**ASCII** code 32) are trimmed.

If the **Any** keyword is used, any character matching a character in *trimset* will be trimmed.

All comparisons are case-sensitive.

Example

```

Dim s1 As String = "Article 101  "
Print "" + RTrim(s1) + ""
Print "" + RTrim(s1, " 01") + ""
Print "" + RTrim(s1, Any " 10") + ""

Dim s2 As String = "Test Pattern aaBBaaBaa"
Print "" + RTrim(s2, "Baa") + ""
Print "" + RTrim(s2, Any "Ba") + ""

```

will produce the output:

```

'Article 101'
'Article 101 '
'Article'
'Test Pattern aaB'
'Test Pattern '

```

Platform Differences

- DOS version/target of FreeBASIC does not support the wide-character version of `RTrim`.

Dialect Differences

- The string type suffix "\$" is obligatory in the *-lang qb* dialect.
- The string type suffix "\$" is optional in the *-lang fblite* and *-lang fb* dialects.

Differences from QB

- QB does not support specifying a *trimset* string or the `ANY` clause.

See also

- LTrim
- Trim

Transfers execution to an external program

Syntax

```
Declare Function Run ( ByRef program As Const String, ByRef argu
```

Usage

```
result = Run( program [, arguments ] )
```

Parameters

program

The file name (including file path) of the program (executable) to trans

arguments

The command-line arguments to be passed to the program.

Return Value

Returns negative one (-1) if the program could not be executed.

Description

Transfers control over to an external program. When the program exit

Example

```
' ' Attempt to transfer control to "program.exe" in  
Dim result As Integer = Run("program.exe")  
  
' ' at this point, "program.exe" has failed to exec  
' ' result will be set to -1.
```

Platform Differences

- Linux requires the *program* case matches the real name of the

insensitive. The program being run may be case sensitive for it

- Path separators in Linux are forward slashes ("/"). Windows uses backslashes ("\\"). Some versions of Windows allow forward slashes. DOS uses backslashes.

Differences from QB

- **Run** needs the full executable name, including extension (.exe) (DOS).
- Returning an error code is new to FreeBASIC.

See also

- **Exec** transfer temporarily, with arguments
- **Chain** transfer temporarily, without arguments
- **Command** pick arguments

SAdd



Returns a pointer to a string variable's data

Syntax

```
Declare Function SAdd ( ByRef str As String ) As ZString Ptr
Declare Function SAdd ( ByRef str As WString ) As ZString Ptr
Declare Function SAdd ( ByRef str As ZString ) As ZString Ptr
```

Usage

```
result = SAdd( str )
```

Parameters

str
the string expression or variable to get the address of

Return Value

A pointer to the data associated with *str*.

Description

Returns the memory offset of the string data in the string variable.

Example

```
Dim s As String

Print SAdd(s)
s = "hello"
Print SAdd(s)
s = "abcdefg, 1234567, 54321"
Print SAdd(s)
```

Differences from QB

- QB returned an integer instead of a pointer.

See also

- [StrPtr](#)
- [VarPtr](#)
- [ProcPtr](#)

Scope...End Scope



Statement to begin a new scope block

Syntax

```
Scope  
  [statements]  
End Scope
```

Description

The Scope block allows variables to be (re)defined and used locally in a program.

When a variable is (re)defined with **Dim** within a scope structure, this local working variable can be used from its (re)definition until the end of the scope. During this time, any variables outside the scope that have the same name will be ignored, and will not be accessible by the name. Any statements in the Scope block before the variable is redefined will use the variable as defined outside the Scope.

Scope . . End Scope is not permitted when compiling with in the *-lang qb* dialect.

Example

```
Dim As Integer x = 5, y = 2  
Print "x ="; x; ", "; "y ="; y  
Scope  
  Dim x As Integer = 3  
  Print "x ="; x; ", "; "y ="; y  
  Scope  
    Dim y As Integer = 4  
    Print "x ="; x; ", "; "y ="; y  
  End Scope  
End Scope  
Print "x ="; x; ", "; "y ="; y
```

Dialect Differences

- Explicit `Scope..End Scope` blocks are available only in the *-lang fb* and *-lang deprecated* dialects.
- Explicit `Scope..End Scope` blocks are not available in the *-lang fb-lite* and *-lang qb* dialects.

Differences from QB

- New to FreeBASIC

See also

- `Dim`
- `ReDim`
- `Static`
- `Var`

Initializes a graphics mode using QB-like mode numbers

Syntax

-lang fb|fb-lite dialects:

```
Screen mode [, [ depth ] [, [ num_pages ] [, [ flags ] [, [ refresh_rate ]  
Screen , [ active_page ] [, [ visible_page ]]
```

-lang qb dialect:

```
Screen [ mode ] [, [ colormode ] [, [ active_page ] [, [ visible_page ]]
```

Parameters

mode

is a QB style graphics screen mode number (see below). If *mode* is 0, the program runs in normal console-mode functionality. See below for available modes.

depth

is the color depth in bits per pixel. This only has an effect for modes 1, 2, and 3. 16 and 32 are aliases for 16 and 32, respectively. If omitted, it defaults to 8.

num_pages

is the number of video pages you want, see below. If omitted, it defaults to 1.

flags

Are used to select several things as graphics driver, fullscreen mode.

ScreenRes for available flags.

refresh_rate

requests a refresh rate. If it is not available in the present card or the driver, it is ignored.

active_page

Used to set the active page, where printing/drawing commands take effect.

visible_page

Used to set the visible page, which is shown to the user.

colormode

Unused - allowed for compatibility with the QB syntax

Description

`screen` tells the compiler to link the GfxLib and initializes a QB-only, QB-style graphics mode.

In QB-only modes a dumb window or fullscreen resolution is set, one redirected to their graphic versions, a **default palette** is set and an auto palette is set.

statements can be used.

In QB-on-GUI modes one or more buffers in standard memory are created. The `palette` is set. QB-like graphics and console statements can be used. graphics buffers.

In OpenGL modes a dumb window or fullscreen resolution is set, one is initialized. From here only OpenGL commands can be used; QB-like is not supported in a portable way; you can then also use `ScreenControl` to properly customize supported OpenGL extensions after a mode has been set, and `Screen`

Any buffer that is created in standard memory uses one of three supported `pixel formats` for details.

If `screen` fails to set the required mode, an "Illegal function call" error is returned. using standard `on Error` processing or retrieving the screen pointer will fail.

Before setting a fullscreen mode the program should check if that mode is supported.

mode details

Available modes list:

QB compatibility modes:

Mode nr	Resolution	Emulation	Text	char size	colors on screen
1	320x200	CGA	40x25	8x8	16 background, 1 of four set
2	640x200	CGA	80x25	8x8	16 colors to 2 attributes
7	320x200	EGA	40x25	8x8	16 colors to 16 attributes
8	640x200	EGA	80x25	8x8	16 colors to 16 attributes
9	640x350	EGA	80x25 Or 80x43	8x14 or 8x8	16 colors to 16 attributes
11	640x480	VGA	80x30 or 80x60	8x16 or 8x8	256K colors to 2 attributes
12	640x480	VGA	80x30 or 80x60	8x16 or 8x8	256K colors to 16 attributes
13	320x200	MCGA	40x25	8x8	256K colors to 256 attributes

New FreeBASIC modes:

Mode nr	Resolution	Emulation	Text	char size	colors on screen
14	320x240		40x30	8x8	256K colors to 256 attrib
15	400x300		50x37	8x8	256K colors to 256 attrib

16	512x384		64x24 or 64x48	8x16 or 8x8	256K colors to 256 attrit
17	640x400		80x25 or 80x50	8x16 or 8x8	256K colors to 256 attrit
18	640x480		80x30 or 80x60	8x16 or 8x8	256K colors to 256 attrit
19	800x600		100x37 or 100x75	8x16 or 8x8	256K colors to 256 attrit
20	1024x768		128x48 or 128x96	8x16 or 8x8	256K colors to 256 attrit
21	1280x1024		160x64 or 160x128	8x16 or 8x8	256K colors to 256 attrit

depth details

For modes 14 and up, the depth parameter changes the color depth to modes 13 and below, *depth* has no effect.

num_pages details

You can request any number of pages for any video mode; if you omit the visible screen or an offscreen buffer, you can show a page while v created in standard memory, the video card memory is never used for

flags details:

(documented at the page [ScreenRes](#))

Other details

While in windowed mode, clicking on the window close button will add window button will switch to fullscreen mode if possible. A successful resets the palette to the specified mode one (see [Default palettes](#)), r mappings, moves the graphics cursor to the center of the screen, mov background colors to bright white and black respectively.

Example

```
' Sets screen mode 13 (320*200, 8bpp)
Screen 13
Print "Screen mode 13 set"

Sleep
```

```

#include "fbgfx.bi"
#if __FB_LANG__ = "fb"
Using FB ' ' Screen mode flags are in the FB namespace
#endif

' Sets screen mode 18 (640*480) with 32bpp color depth
Screen 18, 32, 4, (GFX_WINDOWED Or GFX_NO_SWITCH)

' Check to make sure Screen was opened successfully
If ScreenPtr = 0 Then
    Print "Error setting video mode!"
End
End If

Print "Successfully set video mode"
Sleep

```

Platform Differences

- In DOS, Windowing and OpenGL related switches are not available

Dialect Differences

- In the **-lang fb** and **-lang fbLite** dialects, the usage is:
Screen *mode* [, [*depth*] [, [*num_pages*] [, [*flags*] [, [*refresh_rate*]
Or:
Screen , [*active_page*] [, [*visible_page*]]]

- In the **-lang qb** dialect, the usage is:
Screen [*mode*] [, [*colormode*] [, [*active_page*] [, [*visible_page*]]]

Differences from QB

- None in the **-lang qb** dialect.
- In QB the syntax was **Screen mode, colormode, active_page, visible_page**. The use of **Screen , , apage, vpage** to swap screen pages is not supported.

- `ScreenSet` should be used in the *-lang fb* and *-lang fblite* dialects

See also

- `Screen (Console)`
- `ScreenRes` More flexible alternative to `screen`
- `ScreenList` Check display modes available for FB GfxLib to use
- `ScreenControl` Select driver and more
- `ScreenLock`
- `ScreenUnlock`
- `ScreenPtr` Semi-low level access
- `ScreenSet`
- `ScreenCopy`
- `ScreenInfo`
- `ScreenGLProc`
- `Internal pixel formats`

Screen (Console)



Gets the character or color attribute at a given location

Syntax

```
Declare Function Screen ( ByVal row As Long, ByVal column As Long, colorflag As Long = 0 ) As Long
```

Usage

```
result = Screen( row, column [, colorflag ] )
```

Parameters

row

1-based offset from the top left corner of the console.

column

1-based offset from the top left corner of the console.

colorflag

If equal to 0, the **ASCII** code is returned, otherwise the color attribute omitted, it defaults to 0.

Return Value

The **ASCII** or color attribute of the character.

Description

screen returns the character or the color attribute found at a given position. It works in console mode and in graphics mode.

The format of the color attribute depends on the current color depth:

If the color type is a palette type with up to 4 bits per pixel (such as the **ColorType** type), then the color attribute is an 8-bit value, where the higher four bits hold the background color and the lower four bits hold the foreground (character) color.

If the color type is an 8-bit palette, then the color attribute is a 16-bit value. The high byte holds the background color and the low byte holds the foreground color.

If the color type is full color, then the color attribute is a 32-bit integer, color value. If *colorflag* is equal to 1, then the foreground color is returned. If *colorflag* is equal to 2, then the background color is returned.

The color values for the standard 16 color palette are:

Value	Color	Value	Color
0	Black	8	Gray
1	Blue	9	Bright Blue
2	Green	10	Bright Green
3	Cyan	11	Bright Cyan
4	Red	12	Bright Red
5	Magenta	13	Pink
6	Brown	14	Yellow
7	White	15	Bright White

Example

```
Dim character_ascii_value As Integer
Dim attribute As Integer
Dim background As Integer
Dim cell_color As Integer
Dim row As Integer, col As Integer

character_ascii_value = Screen( row, col )
attribute = Screen( row, col, 1 )
background = attribute Shr 4
cell_color = attribute And &hf
```

```
' ' open a graphics screen with 4 bits per pixel
' ' (alternatively, omit this line to use the console)
ScreenRes 320, 200, 4
```

```

'' print a character
Color 7, 1
Print "A"

Dim As UInteger char, col, fg, bg

'' get the ASCII value of the character we've just
char = Screen(1, 1, 0)

''get the color attributes
col = Screen(1, 1, 1)
fg = col And &HF
bg = (col Shr 4) And &HF

Print Using "ASCII value: ### (!)"; char; Chr(
Print Using "Foreground color: ##"; fg
Print Using "Background color: ##"; bg
Sleep

```

```

'' open a graphics screen with 8 bits per pixel
ScreenRes 320, 200, 8

'' print a character
Color 30, 16
Print "Z"

Dim As UInteger char, col, fg, bg

'' get the ASCII value of the character we've just
char = Screen(1, 1, 0)

''get the color attributes
col = Screen(1, 1, 1)
fg = col And &HFF
bg = (col Shr 8) And &HFF

```

```

Print Using "ASCII value: ### ("!""); char; Chr(
Print Using "Foreground color: ###"; fg
Print Using "Background color: ###"; bg
Sleep

```

```

' ' open a full-color graphics screen
ScreenRes 320, 200, 32

' ' print a character
Color RGB(255, 255, 0), RGB(0, 0, 255) 'yellow on
Print "M"

Dim As Integer char, fg, bg

' ' get the ASCII value of the character we've just
char = Screen(1, 1, 0)

' ' get the color attributes
fg = Screen(1, 1, 1)
bg = Screen(1, 1, 2)

Print Using "ASCII value: ### ("!""); char; Chr(
Print Using "Foreground color: &"; Hex(fg, 8)
Print Using "Background color: &"; Hex(bg, 8)
Sleep

```

Platform Differences

- On the Linux version, the value returned can differ from the character printed on the console. For example, unprintable control codes - such as `chr(10)` that implicitly occurs after the end of **Printed** text - may be printed in place of the untouched character in its place.

Differences from QB

- In QB screen triggered an error if the coordinates were out of s

See also

- [Screen \(Graphics\)](#)
- [Color](#)

Copies the contents of a graphical page into another graphical page

Syntax

```
Declare Function ScreenCopy ( ByVal from_page As Long = -1, ByVal  
to_page As Long = -1 ) As Long
```

Usage

```
ScreenCopy [ from_page ] [, to_page ]
```

Parameters

from_page
page to copy from
to_page
page to copy to

Description

from_page is the page to copy from. If this argument is omitted, the current work page is assumed. *to_page* is the page to copy to. If this argument is omitted, the currently visible page is assumed. Page numbers range from 0 to *num_pages* - 1, where *num_pages* is the number of pages specified when setting the graphics mode with **ScreenRes** or **Screen**.

You can use this function to add a double buffer to your graphics. Any graphics screen mode with multiple pages supports this function.

screenCopy is inactive if the destination page is locked.

There are two other functions similar to this: **Flip** and **PCopy**. **Flip** is designed to work in OpenGL modes, while **PCopy** supports console pages on some platforms. Both do the same thing as **screenCopy** in normal graphics modes.

Example

See also `screenSet` example.

```
' ' 320x200x8, with 3 pages
Screen 13,,3

' ' image for working page #1 (visible page #0)
ScreenSet 1, 0
Cls
Circle( 160, 100 ), 90, 1 ,,,, f
Circle( 160, 100 ), 90, 15
Print "Press 2 to copy page #2 to visible page"
Print "Press escape to exit"

' ' image for working page #2 (visible page #0)
ScreenSet 2, 0
Cls
Line( 50, 50 )-( 270, 150 ), 2, bf
Line( 50, 50 )-( 270, 150 ), 15, b
Print "Press 1 to copy page #1 to visible page"
Print "Press escape to exit"

' ' page #0 is the working page (visible page #0)
ScreenSet 0, 0
Cls
Print "Press 1 to copy page #1 to visible page"
Print "Press 2 to copy page #2 to visible page"
Print "Press escape to exit"

Dim k As String

Do
    k = Inkey
    Select Case k
    Case Chr(27)
        Exit Do
    Case "1"
        ScreenCopy 1, 0
    Case "2"
```

```
ScreenCopy 2, 0
End Select

Sleep 25
Loop
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__ScreenCopy`.

Differences from QB

- New to FreeBASIC. It is a graphics-only version of `PCopy` - which works in both text and graphics modes.

See also

- `PCopy`
- `Screen (Graphics)`
- `ScreenRes`
- `ScreenSet`

ScreenControl



Sets or gets internal graphics library settings

Syntax

```
Declare Sub ScreenControl ( ByVal what As Long, ByRef param1 As Integer = 0, ByRef param3 As Integer = 0, ByRef param4 As Integer = 0 )
Declare Sub ScreenControl ( ByVal what As Long, ByRef param As String )
```

Usage

```
ScreenControl( what [, [ param1 ][, [ param2 ][, [ param3 ][, [ param4 ]]
```

```
ScreenControl( what [, param ] )
```

Parameters

what

specifies the function to perform

param1

optional first integer parameter, contains value to be set on entry or value returned on exit

param2

optional second integer parameter, contains value to be set on entry or value returned on exit

param3

optional third integer parameter, contains value to be set on entry or value returned on exit

param4

optional fourth integer parameter, contains value to be set on entry or value returned on exit

param

optional string parameter, contains text to be set on entry or text got on exit

Description

This function can be used to set or get internal GfxLib states. The *what* parameter specifies the function to perform. On operations that set states, the *param** parameters must contain the values to be set. On operations that get states, *param** will hold the values returned by GfxLib. The meaning of the *param** parameters depend on the *what* parameter, and are defined by the constants in `fbgfx.bi`. In **lang fb**, they are set to be stored in the `FB Nam`. Below is a list of the supported *what* constants - and their values as defined in `fbgfx.bi` and the parameters associated with them.

Supported operations

Note: * denotes operations that are allowed while a graphics mode has **(Graphics)** or **ScreenRes**. For all other operations, return values are zero. An operation has no effect if a graphics mode is not available at call time.

Get operations

- GET_WINDOW_POS (0) Returns the current window position, in desktop coordinates.

[OUT] *param1* X

[OUT] *param2* Y

- * GET_WINDOW_TITLE (1) Returns the title of the program window.

[OUT] *param* title

- GET_WINDOW_HANDLE (2) Returns a handle to the program window.

[OUT] *param1* handle; this is a HWND in Windows, a "Window" XID in X11

- * GET_DESKTOP_SIZE (3) Returns the desktop size, in pixels.

[OUT] *param1* width

[OUT] *param2* height

- GET_SCREEN_SIZE (4) Returns the current screen size in pixels.

[OUT] *param1* width

[OUT] *param2* height

- GET_SCREEN_DEPTH (5) Returns current graphics mode screen depth.

[OUT] *param1* bits per pixel

- GET_SCREEN_BPP (6) Returns current graphics mode BPP.

[OUT] *param1* bytes per pixel

- GET_SCREEN_PITCH (7) Returns the current graphics mode framebuffer pitch.

[OUT] *param1* pitch

- GET_SCREEN_REFRESH (8) Returns the current graphics mode refresh rate.

[OUT] *param1* rate

- GET_DRIVER_NAME (9) Returns the current graphics mode driver name.

[OUT] *param* name

- GET_TRANSPARENT_COLOR (10) Returns the transparent color value for the current graphics mode.

[OUT] *param1* value

- GET_VIEWPORT (11) Returns the current viewport as set by the **ViewPort** coordinates.

[OUT] *param1* X1

[OUT] *param2* Y1

[OUT] *param3* x2

[OUT] *param4* y2

- GET_PEN_POS (12) Returns the last graphical pen position, in screen coordinates, in graphics functions supporting relative coordinates using the [Step](#)

[OUT] *param1* X

[OUT] *param2* y

- GET_COLOR (13) Returns the current graphics mode color.

[OUT] *param1* foreground

[OUT] *param2* background

- GET_ALPHA_PRIMITIVES (14) Returns if primitives drawing support is enabled

[OUT] *param1* TRUE (-1) if alpha primitives is enabled, FALSE (0) otherwise

- GET_GL_EXTENSIONS (15) Returns a string holding all supported GL extensions in OpenGL mode.

[OUT] *param* supported GL extensions

- GET_HIGH_PRIORITY (16) Returns if GFX_HIGH_PRIORITY was specified in [ScreenRes](#).

[OUT] *param1* higher priority graphics processing enabled

Set operations

- SET_WINDOW_POS (100) Sets the current program window position,

[IN] *param1* X

[IN] *param2* y

- * SET_WINDOW_TITLE (101) Sets the current program window title. 101 [WindowTitle\(param \)](#).

[IN] *param* title

- SET_PEN_POS (102) Sets the current graphical pen position, in screen coordinates, in graphics functions supporting relative coordinates using the [Step](#)

[IN] *param1* X

[IN] *param2* y

- * SET_DRIVER_NAME (103) Sets the name of the internal graphics driver. 103 [Screen](#) Or [ScreenRes](#).

[IN] *param* driver name

- SET_ALPHA_PRIMITIVES (104) Sets if primitives drawing should honor alpha

[IN] *param1* enabled

- * SET_GL_COLOR_BITS (105) Sets the number of bits dedicated to the color

[IN] *param1* bits

- * SET_GL_COLOR_RED_BITS (106) Sets the number of bits dedicated color buffer

[IN] *param1* bits

- * SET_GL_COLOR_GREEN_BITS (107) Sets the number of bits dedicated OpenGL color buffer

[IN] *param1* bits

- * SET_GL_COLOR_BLUE_BITS (108) Sets the number of bits dedicated color buffer

[IN] *param1* bits

- * SET_GL_COLOR_ALPHA_BITS (109) Sets the number of bits dedicated OpenGL color buffer

[IN] *param1* bits

- * SET_GL_DEPTH_BITS (110) Sets the number of bits dedicated to the

[IN] *param1* bits

- * SET_GL_STENCIL_BITS (111) Sets the number of bits dedicated to the

[IN] *param1* bits

- * SET_GL_ACCUM_BITS (112) Sets the number of bits dedicated to the

[IN] *param1* bits

- * SET_GL_ACCUM_RED_BITS (113) Sets the number of bits dedicated accumulation buffer

[IN] *param1* bits

- * SET_GL_ACCUM_GREEN_BITS (114) Sets the number of bits dedicated OpenGL accumulation buffer

[IN] *param1* bits

- * SET_GL_ACCUM_BLUE_BITS (115) Sets the number of bits dedicated accumulation buffer

[IN] *param1* bits

- * SET_GL_ACCUM_ALPHA_BITS (116) Sets the number of bits dedicated OpenGL accumulation buffer

[IN] *param1* bits

- * SET_GL_NUM_SAMPLES (117) Sets the number of samples to be used

[IN] *param1* samples

Other operations

- POLL_EVENTS (200) Cause the library to poll all events, ie to check used for retrieving keyboard and mouse events. This is most used

used, as normally `Flip` will cause these events to be polled.

Example

```
' ' include fbgfx.bi for some useful definitions
#include "fbgfx.bi"

' ' use FB namespace for easy access to types/consta
Using FB

Dim e As EVENT
Dim As Integer x0, y0, x, y
Dim As Integer shakes = 0
Dim As Any Ptr img

ScreenRes 320, 200, 32
Print "Click to shake window"

' ' find window coordinates
ScreenControl GET_WINDOW_POS, x0, y0

Do

    If (shakes > 0) Then

        ' ' do a shake of the window

        If (shakes > 1) Then

            ' ' move window to a random position near
            x = x0 + Int(32 * (Rnd() - 0.5))
            y = y0 + Int(32 * (Rnd() - 0.5))
            ScreenControl SET_WINDOW_POS, x, y

        Else

            ' ' move window back to its original coo
            ScreenControl SET_WINDOW_POS, x0, y0

        End If

    End If

End Do
```

```

        End If

        shakes -= 1

    End If

    If (ScreenEvent(@e)) Then
        Select Case e.type

            '' user pressed the mouse button
            Case EVENT_MOUSE_BUTTON_PRESS

                If (shakes = 0) Then
                    '' set to do 20 shakes
                    shakes = 20

                    '' find current window coordinates
                    ScreenControl GET_WINDOW_POS, x0, y
                End If

                '' user closed the window or pressed a key
                Case EVENT_WINDOW_CLOSE, EVENT_KEY_PRESS
                    '' exit to end of program
                    Exit Do

            End Select
        End If

        '' free up CPU for other programs
        Sleep 5

    Loop

```

```

'' include fbgfx.bi for some useful definitions
#include "fbgfx.bi"

```

```

Dim As String driver

#ifdef __FB_WIN32__
' set graphics driver to GDI (Win32 only), before
ScreenControl FB.SET_DRIVER_NAME, "GDI"
#endif

ScreenRes 640, 480

' fetch graphics driver name and display it to use
ScreenControl FB.GET_DRIVER_NAME, driver
Print "Graphics driver name: " & driver

' wait for a keypress before closing the window
Sleep

```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the *z*

Differences from QB

- New to FreeBASIC

See also

- [Screen \(Graphics\)](#)
- [ScreenEvent](#)
- [ScreenInfo](#)
- [WindowTitle](#)
- [View \(Graphics\)](#)

ScreenEvent



Queries for and retrieves system events.

Syntax

```
Declare Function ScreenEvent ( ByVal event As Any Ptr = 0 ) As L
```

Usage

```
result = ScreenEvent( [ event ] )
```

Parameters

event

Specifies the buffer where the function should store the event data.

Return Value

Returns -1 if there are pending events to be retrieved, 0 otherwise.

Description

This function returns the latest available system event from the internal keyboard activity, for example.

The event data (if available) will be copied into the buffer pointed That

Querying for events

The function returns -1 if there are pending events to be retrieved, 0 0 **ScreenEvent** will not be able to copy the event data and it will not dequ way can be useful to check if there are pending events without actual

Note

If you receive a KEY_PRESS, KEY_RELEASE or KEY_REPEAT event be clear after you receive the event, you will need to clear it manually.

Example

```

'' include fbgfx.bi for some useful definitions
#include "fbgfx.bi"
#if __FB_LANG__ = "fb"
Using fb '' constants and structures are stored in
#endif

Dim e As EVENT

ScreenRes 640, 480
Do
    If (ScreenEvent(@e)) Then
        Select Case e.type
            Case EVENT_KEY_PRESS
                If (e.scancode = SC_ESCAPE) Then
                    End
                End If
                If (e.ascii > 0) Then
                    Print "" & e.ascii & "";
                Else
                    Print "unknown key";
                End If
                Print " was pressed (scancode " & e.sc
            Case EVENT_KEY_RELEASE
                If (e.ascii > 0) Then
                    Print "" & e.ascii & "";
                Else
                    Print "unknown key";
                End If
                Print " was released (scancode " & e.s
            Case EVENT_KEY_REPEAT
                If (e.ascii > 0) Then
                    Print "" & e.ascii & "";
                Else
                    Print "unknown key";
                End If
                Print " is being repeated (scancode "
            Case EVENT_MOUSE_MOVE
                Print "mouse moved to " & e.x & ", " &

```

```

Case EVENT_MOUSE_BUTTON_PRESS
  If (e.button = BUTTON_LEFT) Then
    Print "left";
  ElseIf (e.button = BUTTON_RIGHT) Then
    Print "right";
  Else
    Print "middle";
  End If
  Print " button pressed"
Case EVENT_MOUSE_BUTTON_RELEASE
  If (e.button = BUTTON_LEFT) Then
    Print "left";
  ElseIf (e.button = BUTTON_RIGHT) Then
    Print "right";
  Else
    Print "middle";
  End If
  Print " button released"
Case EVENT_MOUSE_DOUBLE_CLICK
  If (e.button = BUTTON_LEFT) Then
    Print "left";
  ElseIf (e.button = BUTTON_RIGHT) Then
    Print "right";
  Else
    Print "middle";
  End If
  Print " button double clicked"
Case EVENT_MOUSE_WHEEL
  Print "mouse wheel moved to position "
Case EVENT_MOUSE_ENTER
  Print "mouse moved into program window"
Case EVENT_MOUSE_EXIT
  Print "mouse moved out of program window"
Case EVENT_WINDOW_GOT_FOCUS
  Print "program window got focus"
Case EVENT_WINDOW_LOST_FOCUS
  Print "program window lost focus"
Case EVENT_WINDOW_CLOSE
  End

```

```
        Case EVENT_MOUSE_HWHEEL
            Print "horizontal mouse wheel moved to"
        End Select
    End If

    Sleep 1
Loop
```

Platform Differences

- ScreenEvent does not return window related events in the DOS

Dialect Differences

- Not available in the *-lang qb* dialect.

Differences from QB

- New to FreeBASIC

See also

- Event
- Screen (Graphics)
- Inkey
- MultiKey
- GetMouse

h	Height of the screen in pixels
depth	Current pixel format bits per pixel: this can be 1, 2, 4, 8, 16, or 32
pitch	Size of a framebuffer row in bytes
rate	Current refresh rate, or 0 if unknown
driver	Name of current graphics driver in use, like DirectX or X11

Example

```

Dim w As Integer, h As Integer
Dim depth As Integer
Dim driver_name As String

Screen 15, 32
' Obtain info about current mode
ScreenInfo w, h, depth,,,driver_name
Print Str(w) + "x" + Str(h) + "x" + Str(depth);
Print " using " + driver_name + " driver"
Sleep
' Quit graphics mode and obtain info about desktop
Screen 0
ScreenInfo w, h, depth
Print "Desktop running at " + Str(w) + "x" + Str(h)

```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- [Screen \(Graphics\)](#)

ScreenGLProc



Gets the address of an OpenGL procedure

Syntax

```
Declare Function ScreenGLProc ( ByRef procname As Const String )
```

Parameters

procname

name of the procedure to retrieve the address of

Description

This function can be used to get the address of any OpenGL procedure functions associated with OpenGL extensions. If given procedure name will return NULL (0).

Example

```
' ' include fbgfx.bi for some useful definitions
#include "fbgfx.bi"

Dim SwapInterval As Function(ByVal interval As Int)
Dim extensions As String

' ' Setup OpenGL and retrieve supported extensions
ScreenRes 640, 480, 32,, FB.GFX_OPENGL
ScreenControl FB.GET_GL_EXTENSIONS, extensions

If (InStr(extensions, "WGL_EXT_swap_control") <> 0) Then
    ' ' extension supported, retrieve proc address
    SwapInterval = ScreenGLProc("wglSwapIntervalEXT")
    If (SwapInterval <> 0) Then
        ' ' Ok, we got it. Set OpenGL to wait for v
        SwapInterval(1)
    End If
End If
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Platform Differences

- Not available for DOS target.

Differences from QB

- New to FreeBASIC

See also

- [Screen \(Graphics\)](#)
- [ScreenControl](#)

ScreenList



Finds available fullscreen video modes

Syntax

```
Declare Function ScreenList ( ByVal depth As Long = 0 ) As Long
```

Usage

```
result = ScreenList( [ depth ] )
```

Parameters

depth

the color depth for which the list of modes is requested (supported depths are 8, 15, 16, 24 and 32)

Return Value

returns 0, when there are no more resolutions to read.

Description

It works like the **dir** function: the first call to the function requires the *depth* parameter to be specified, it returns the lowest supported resolution for the requested depth. Further calls to **ScreenList** without arguments return the next resolutions. When no more resolutions are available, **ScreenList** returns 0.

The result of **ScreenList** is encoded as a 32 bit value, with the screen width as the **High Word** and the height as the **Low Word**.

Resolutions are returned from lowest to highest supported ones.

It is safe to call this function before any graphics mode has been set.

```
Dim As Integer mode, w, h
```

```
Print "Resolutions supported at 8 bits per pixel:"  
  
mode = ScreenList(8)  
While (mode <> 0)  
    w = HiWord(mode)  
    h = LowWord(mode)  
    Print w & "x" & h  
    mode = ScreenList()  
Wend
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the `__ScreenList`.

Differences from QB

- New to FreeBASIC

See also

- [Screen](#)
- [ScreenRes](#)

Locks the working page's frame buffer

Syntax

```
Declare Sub ScreenLock ( )
```

Usage

```
ScreenLock
```

Description

All of FreeBASIC's Graphics Library functions draw to a frame buffer and an automatic routine copies the frame buffer to the actual screen memory at each draw. If the user program does a lot of drawing, the automatic refreshes may take a significant amount of time.

The **ScreenLock** function locks the automatic refresh, so several drawing operations may be done before the screen refresh is performed, thus increasing the speed of execution, and preventing the user from seeing partial results.

Frame buffer memory may be freely accessed by using pointers (see **ScreenPtr**) ONLY while the screen is locked. Primitive graphics statements (**Line**, **PSet**, **Draw String**, ...) may be used at any time.

The screen refresh remains locked until the use of **ScreenUnlock** statement, which resumes it.

Calls to **ScreenLock** must be paired with a matching call to **ScreenUnlock**. The graphics driver keeps track of how many times **ScreenLock** has been called using a counter. Only the first call to **ScreenLock** actually performs a locking operation. Subsequent calls to **ScreenLock** only increment the counter. Conversely, **ScreenUnlock** only decrements the lock counter until it reaches zero at which time the actual unlock operation will be performed. Using **Screen** or **ScreenRes** will release all locks and set the

lock counter back to zero before changing screen modes.

It is strongly recommended that the lock on a page be held for as short time as possible. Only screen drawing should occur while the screen is locked, input/output and waiting must be avoided. In Win32 and Linux screen is locked by stopping the thread that processes also the OS' events. If the screen is kept locked for a long time the event queue can overflow and make the system unstable. When the induced lock time becomes too long, use preferably the method of double buffering (with `ScreenCopy`).

The automatic refresh takes place only in the visible page of the frame buffer. `ScreenLock` has no effect when drawing to pages other than the visible one.

Example

```
' ' Draws a circle on-screen at the mouse cursor
Dim As Integer mx, my
Dim As String key

ScreenRes 640, 480, 32

Do

    'process
    GetMouse(mx, my)
    key = Inkey()

    'draw
    ScreenLock()
    Cls()
    Circle (mx, my), 8, RGB(255, 255, 255)
    ScreenUnlock()

    'free up CPU time
    Sleep(18, 1)
```

```
Loop Until key = Chr(27) Or key = Chr(255, 107)
```

Platform Differences

- In DOS, the mouse arrow does not react to mouse movements while the screen is locked

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__ScreenLock`.

Differences from QB

- New to FreeBASIC

See also

- `Screen (Graphics)` - Setting mode
- `ScreenRes` - Setting mode
- `ScreenUnlock`
- `ScreenPtr`

ScreenPtr



Returns a pointer to the current work page's frame buffer

Syntax

```
Declare Function ScreenPtr ( ) As Any Ptr
```

Usage

```
result = ScreenPtr
```

Return Value

a pointer to the current work page frame buffer memory, or NULL (0) if

Description

`ScreenPtr` provides a way to directly read/write the working page's frame buffer. Any read or writes are attempted. The pointer returned is valid up until the next call to `ScreenPtr`, which invalidates it.

`ScreenPtr` can also be used to test if a call to `Screen` or `ScreenRes` was successful. The return value is non-zero if successful.

In order to access a pixel in the screen buffer, you will need to know the row and column (y and x), and also the width and height to avoid going out of bounds. This is done using the `ScreenInfo` structure.

Each row in the frame buffer is *pitch* bytes long. The frame buffer coordinates are given in terms of position on the screen, running from top to bottom, left to right.

Because of the design of FreeBASIC graphics library, `ScreenPtr` (if non-zero) points to actual video RAM.

Example

```
Const SCREEN_WIDTH = 640, SCREEN_HEIGHT = 480
Dim As Integer w, h, bypp, pitch
```

```

'' Make 8-bit screen.
ScreenRes SCREEN_WIDTH, SCREEN_HEIGHT, 8

'' Get screen info (w and h should match the const
ScreenInfo w, h, , bypp, pitch

'' Get the address of the frame buffer. An Any Ptr
'' is used here to allow simple pointer arithmetic
Dim buffer As Any Ptr = ScreenPtr()
If (buffer = 0) Then
    Print "Error: graphics screen not initialized."
    Sleep
    End -1
End If

'' Lock the screen to allow direct frame buffer access
ScreenLock()

    '' Find the address of the pixel in the centre
    '' It's an 8-bit pixel, so use a UByte Ptr.
    Dim As Integer x = w \ 2, y = h \ 2
    Dim As UByte Ptr pixel = buffer + (y * pitch)

    '' Set the pixel color to 10 (light green).
    *pixel = 10

'' Unlock the screen.
ScreenUnlock()

'' Wait for the user to press a key before closing
Sleep

```

```

Const SCREEN_WIDTH = 256, SCREEN_HEIGHT = 256
Dim As Integer w, h, bypp, pitch

```

```

'' Make 32-bit screen.

```

```

ScreenRes SCREEN_WIDTH, SCREEN_HEIGHT, 32

'' Get screen info (w and h should match the const
ScreenInfo w, h, , bypp, pitch

'' Get the address of the frame buffer. An Any Ptr
'' is used here to allow simple pointer arithmetic
Dim buffer As Any Ptr = ScreenPtr()
If (buffer = 0) Then
    Print "Error: graphics screen not initialized."
    Sleep
    End -1
End If

'' Lock the screen to allow direct frame buffer access
ScreenLock()

'' Set row address to the start of the buffer
Dim As Any Ptr row = buffer

'' Iterate over all the pixels in the screen:
For y As Integer = 0 To h - 1

    '' Set pixel address to the start of the row
    '' It's a 32-bit pixel, so use a ULong Ptr
    Dim As ULong Ptr pixel = row

    For x As Integer = 0 To w - 1

        '' Set the pixel value
        *pixel = RGB(x, x Xor y, y)

        '' Get the next pixel address
        '' (ULong Ptr will increment by 4 bytes)
        pixel += 1

    Next x

```

```
    ' ' Go to the next row
    row += pitch

    Next y

' ' Unlock the screen.
ScreenUnlock()

' ' Wait for the user to press a key before closing
Sleep
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- [Screen \(Graphics\)](#)
- [ScreenRes](#)
- [ScreenInfo](#)
- [ScreenLock](#)
- [ScreenUnlock](#)

ScreenRes



Initializes a graphics mode by specifying horizontal and vertical resolution

Syntax

```
Declare Function ScreenRes ( ByVal width As Long, ByVal height As Long,
ByVal num_pages As Long = 1, ByVal flags As Long = 0, ByVal refresh_rate As Long = 0 ) As Long
```

Usage

```
ScreenRes width, height [, [depth] [, [num_pages] [, [flags] [, [refresh_rate]
result = ScreenRes( width, height [, [depth] [, [num_pages] [, [flags] [, [refresh_rate]
```

Parameters

width, height

The display width and height, respectively. For fullscreen mode, the user must specify the resolution using [ScreenList](#).

depth

The color depth in bits per pixel. Valid color depths are: 1, 2, 4, 8, 16 and 32. 16 and 32 are allowed as aliases for 16 and 32, respectively. If omitted, the default is 16. If 1 is specified, it will give a palette image. The default palette will be the first 2^{depth} colors in the palette.

[Screen](#) 13.

num_pages

The number of video pages to create, defaults to 1. (see below)

flags

Used to set various properties of the screen, including fullscreen mode. See the [Screen](#) header below or the standard header "fbgfx.bi" for available flags)

refresh_rate

The desired refresh rate of the screen, only has an effect for fullscreen mode. Defaults to an appropriate value, invalid refresh rates will be ignored.

Return Value

Returns zero (0) if successful, or a non-zero error code to indicate a failure.

Description

ScreenRes tells the compiler to link the GfxLib and initializes a QB-only depending on the *flags* setting

ScreenRes clears the created window or the full screen. In non-fullscreen to match any resolution of the graphics card. Resolutions like 555x111 window of such size. See the page [GfxLib overview](#) for DOS issues.

The font size in **ScreenRes** modes is set to 8x8 by default. This can be rows/columns, using the [width](#) function.

In QB-only modes a dumb window or fullscreen resolution is set, one created, console commands are redirected to their graphic versions, a screen refresh thread is started. QB-like graphics and console statements

In QB-on-GUI modes one or more buffers in standard memory are created to their graphic versions and a [default palette](#) is set. QB-like graphics is up to the user to create a window and to refresh it with the contents

In OpenGL modes a dumb window or fullscreen resolution is set, one created, and the system's OpenGL library is initialized. From here only write to the graphics buffer. QB-like and console commands are forbidden. OpenGL in a portable way.

flags details:

If flags are omitted, FreeBASIC uses QB-compatible graphics in window constants are defined in `fbgfx.bi`. In the *-lang fb* dialect, these constant values can be combined to form a mask using [operator Or](#). Note that DOS.

Available flags:

graphic mode flags

GFX_NULL: Starts a QB-on-GUI graphics mode. It creates a graphics buffer, implements the window, the events manager and refreshes the screen as FreeBASIC drawing functions with API-driven windows. Alternatively, example files) without making it visible on the screen, even in a purely all other mode flags. See an [Example of GFX_NULL](#) in Windows.

GFX_OPENGL: Initializes OpenGL to draw in a dumb window. FreeBASIC

screen is not automatically updated, **Flip** must be used. This option p
OpenGL Library.

If none of the above options is specified, FreeBASIC enters the QB-or
and a dumb window and sets a thread that automatically updates the
mouse. The FreeBASIC drawing functions can be used.

window mode flags

Window mode flags are meaningless if `GFX_NULL` mode is used

`GFX_WINDOWED`: If windowed mode is supported, FreeBASIC opens a wi
present desktop

`GFX_FULLSCREEN`: The graphics card switch mode is switched to the req
fullscreen mode is used. If the mode is not available in the present ca
mode.

If `GFX_FULLSCREEN` is not specified, the behavior for `GFX_WINDOWED` is ass

`GFX_NO_SWITCH`: Prevents the user from changing to fullscreen or to wir

`GFX_NO_FRAME`: Creates a window without a border.

`GFX_SHAPED_WINDOW`: Creates transparent regions wherever `RGBA(255`

`GFX_ALWAYS_ON_TOP`: Creates a window that stays always on top.

option flags

Flags working in any mode, they activate special behaviors

`GFX_ALPHA_PRIMITIVES`: Tells the graphics library to enable alpha chanr

This means the alpha specified in a color value (via either the `RGBA` ma
&h;AARRGGBB) will always be used by all primitives.

`GFX_HIGH_PRIORITY`: Tells the graphics library to enable a higher priority
effect on `gdi` and `DirectX` drivers on Win32 platform.

OpenGL Buffer flags

These flags work only in OpenGL graphics mode, must be combined v

`GFX_STENCIL_BUFFER`: Forces OpenGL to use Stencil buffer

`GFX_ACCUMULATION_BUFFER`: Forces OpenGL to use Accumulation buffer

`GFX_MULTISAMPLE`: Requests fullscreen anti-aliasing through the `ARB_r`

Depending on whether the `GFX_FULLSCREEN` parameter is present or no
video mode in fullscreen or windowed mode, respectively. If fullscree
specified mode in fullscreen, it will try in windowed mode. If windowed
open a window for specified mode, it will try fullscreen. If everything fa
execution will resume from the statement following the `screen` call. Yo
graphics mode has been set or not, and behave accordingly; a way to

the return value of the `ScreenPtr` function; see its page for details.

Graphics mode console

Console commands (`Locate`, `Print`), input can be used both with standard and extended ones too, provided the standard color depth is not modified. Where the table says more than one text resolution is available for the console, the one to be requested can be requested by using `width`. Any characters `Printed` will erase the background. You can use a transparent background.

Example

```
' Set the screen mode to 320*200, with 8 bits per
ScreenRes 320, 200, 8

' Draw color bands in a diagonal pattern over the
For y As Integer = 0 To 200-1
    For x As Integer = 0 To 320-1
        PSet (x,y),(x + y) And 255
    Next x
Next y

' Display the text "Hello World!!" over the lines
left hand corner
Print "Hello world!!"

' Keep the window open until the user presses a key
Sleep
```

Platform Differences

- In DOS, Windowing and OpenGL related switches are not available. See [overview](#)

Dialect Differences

- Not available in the `-lang qb` dialect unless referenced with the `ScreenPtr` function.

Differences from QB

- New to FreeBASIC

See also

- **Screen** The QB-like way to set graphics mode
- **ScreenList** Check display modes available for FB GfxLib to use
- **ScreenControl** Select driver and more
- **ScreenLock**
- **ScreenUnlock**
- **ScreenPtr** Semi-low level access
- **ScreenSet**
- **ScreenCopy**
- **ScreenInfo**
- **ScreenGLProc**
- **Internal pixel formats**
- **FaqPggfxlib2**

ScreenSet



Sets current work and visible pages

Syntax

```
Declare Sub ScreenSet ( ByVal work_page As Long = -1, ByVal visi
```

Usage

```
ScreenSet [ work_page ] [, visible_page ]
```

Parameters

work_page
index to working page
visible_page
index to visible page

Description

`screenSet` allows to set the current working page and the current visible page. The number of pages specified when setting the graphics mode with `ScreenRes` and `ScreenSet` is used for double-buffering.

If you provide *visible_page* but omit *work_page*, only the visible page is changed. If you omit both arguments, both work page and visible page is changed.

`screenSet` provides one method of writing to the screen without instantiating a `Screen` object. An alternative method of doing this is `Screen`.

Example

```
' Open graphics screen (320*200, 8bpp) with 2 pages
ScreenRes 320, 200, 8, 2

' Work on page 1 while displaying page 0
ScreenSet 1, 0
```

```
Dim As Integer x = -40

Do
    ' Clear the screen, draw a box, update x
    Cls
    Line (x, 80)-Step(39, 39), 4, BF
    x += 1: If (x > 319) Then x = -40

    ' Wait for vertical sync: only used to control
    ScreenSync

    ' Copy work page to visible page
    ScreenCopy

Loop While Inkey = ""
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- [Screen \(Graphics\)](#)
- [ScreenRes](#)
- [ScreenCopy](#)
- [ScreenLock](#)
- [ScreenUnlock](#)

Synchronizes display updates with hardware

Syntax

Declare Function ScreenSync () As Long

Usage

result = **ScreenSync**

Return Value

Zero if successful, or non-zero if a graphics mode was not previously set.

Description

This GfxLib statement stops the execution of the program until the graphics card signals it has ended tracing a frame and is going to start the new one.

If the program uses this small interval of time between frames to redraw the image, the flickering is greatly reduced. In that use, **ScreenSync** is a reminiscence of QB where there was only that equivalent method (**wait** &H3DA;, 8) to improve the flickering. It is an empirical method because it only allows to synchronize the beginning of the drawing with the fixed dead time between two frames. To be used occasionally to avoid flickering when only very short time of drawing.

Except the purpose to reduce the flickering, **ScreenSync** can be also used simply as a method of synchronization of graphic drawing with the screen frame tracing (similarly to statement **sleep**).

The use of the QB-compatible form **wait** &H3DA;, 8 is deprecated.

Example

```
'main loop
Do

    ' do user input
    ' calculate_a_frame

    ScreenSync

    ' draw_ a_ frame

Loop Until Inkey <> ""
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__Screensync`.

Differences from QB

- New to FreeBASIC.
- QBasic used `wait &H3DA;, 8` for this purpose.

See also

- `wait`

ScreenUnlock



Unlocks work page's framebuffer

Syntax

```
Declare Sub ScreenUnlock ( ByVal startline As Long = -1, ByVal  
    endlire As Long = -1 )
```

Usage

```
ScreenUnlock [ start_line ] [, end_line ]
```

Parameters

startline

optional argument specifying first screen line to be updated. If omitted top screen line is assumed.

endlire

optional argument specifying last screen line to be updated. If omitted bottom screen line is assumed.

Description

ScreenUnlock unlocks the current work page assuming it was previously locked by calling **ScreenLock** and lets the system restart updating the screen regularly. When called with *start_line* and *end_line* , only the screen area between those lines is assumed to have changed, and will be updated.

An internal counter exists that remembers the screen lock state, thus **ScreenUnlock** has an effect only on a screen that is locked. A screen that has not been locked with **ScreenLock** cannot get unlocked, however **ScreenUnlock** still will force an update of given area or full screen.

Calls to **ScreenUnlock** must be paired with matching calls to **ScreenLoc** Only the first call to **ScreenLock** actually performs a locking operation. Subsequent calls to **ScreenLock** only increment the lock counter. Conversely, **ScreenUnlock** only decrements the lock counter until it

reaches zero at which time the actual unlock operation will be performed. Using `Screen` or `ScreenRes` will release all locks and set the lock counter back to zero before changing screen modes.

All graphic statements automatically lock the screen before the function call, and unlock the screen afterwards, so you do not need to do this explicitly using `ScreenLock` and `ScreenUnlock`. You only need to lock the screen when you wish to access the screen (framebuffer) directly using `ScreenPtr` or when you wish to group several graphic statements together so their effects appear simultaneously on screen, thus avoiding potential screen flicker during screen updates.

Warning (Win32, Linux) : The screen is locked by stopping the thread that processes also the OS' events. This means the screen should be locked only for the short time required to redraw it, and no user input will be received while the screen is locked. When the induced lock time becomes too long, use preferably the method of double buffering (with `ScreenCopy`).

Example

See `ScreenPtr` example.

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__Screenunlock`.

Differences from QB

- New to FreeBASIC

See also

- `Screen (Graphics)`
- `ScreenLock`
- `ScreenPtr`

Second



Gets the seconds from a **Date Serial**

Syntax

```
Declare Function Second ( ByVal date_serial As Double ) As Long
```

Usage

```
#include "vbcompat.bi"  
result = Second( date_serial )
```

Parameters

date_serial
the date serial

Return Value

Returns the seconds from a variable containing a date in **Date Serial**

Description

The compiler will not recognize this function unless `vbcompat.bi` is inc

Example

```
#include "vbcompat.bi"  
  
Dim ds As Double = DateSerial(2005, 11, 28) + Time  
Print Format(ds, "yyyy/mm/dd hh:mm:ss "); Second(c
```

Differences from QB

- Did not exist in QB. This function appeared in PDS and VBDO:

See also

- **Date Serials**

Seek (Statement)



Sets the position of the next read/write operation on a file

Syntax

```
Seek [#]filenum, position
```

Parameters

filenum

file number of an opened a file

position

the new position for i/o operations

Description

Sets the position at which the next read or write operation on a file will

The position is given in records if the file was opened in **Random** access: position is 1 based -- the first record of a file is at position 1.

The **seek** function is used to get the position of the next read or write c

Example

```
' e.g. if you want to skip to the 100th byte in th  
  
Dim f As Integer  
  
f = FreeFile  
Open "file.ext" For Binary As #f  
  
Seek f, 100  
  
Close #f
```

Differences from QB

- None

See also

- [Seek \(Function\)](#)
- [Open](#)

Seek (Function)



Gets the position of the next read/write operation for a file or device

Syntax

```
Declare Function Seek ( ByVal filenum As Long ) As LongInt
```

Parameters

filenum
file number of an open file

Return Value

The file position where the next read or write operation will take place

Description

The position is given in records if the file was opened in **Random** access mode, in bytes in any other case. The file position returned is 1-based so the first record of a file is 1.

The **seek** statement is used to set the position of the next read or write operation.

Example

```
Dim f As Integer, position As Integer

f = FreeFile
Open "file.ext" For Binary As #f

position = Seek(f)

Close #f
```

Differences from QB

- None

See also

- [Seek \(Statement\)](#)
- [LOC](#)
- [Open](#)

Select Case



Conditional statement block

Syntax

```
Select Case expression
[ Case expressionlist ]
[ statements ]
[ Case Else ]
[ statements ]
End Select
or
Select Case As Const integer_expression
[ Case constant | enumeration ]
[ statements ]
[ Case Else ]
[ statements ]
End Select
```

Description

select case executes specific code depending on the value of an expression. The expression is evaluated once, and compared against each **case**, in order, until a match is found. The code inside the matching **Case** branch is executed, and then the program continues at the end of the **select case** block. **case Else** matches any case not already matched. If no **case** matches, at least one **case** is guaranteed to be executed. If no **case** matches, the **select case** block will be skipped.

End select is used to close the **select case...End select** block.

Note for C users: In FreeBASIC, **select case** works like a **switch** block. There is no fall-through, multiple options must be listed in a single **case**.

Besides integer types, floating point and string expressions are also supported in the **select case** syntax.

Syntax of an expression list:

```
{ expression | expression To expression | Is relational operator }
```

- *expr*: evaluates *expr*, and compares for equality with the original expression. If equal, then a match has been found. This could be considered "is" (see below).
- *expr1 To expr2*: evaluates *expr1* and checks to see if it is less than or equal to the original expression. If so, it evaluates *expr2*, and checks to see if it is equal to the original expression. If so, then a match has been found.
- **Is** *relational_operator expr*: evaluates *expr*, and compares it against it, using the supplied *relational_operator* (=, >, <, <>, <=, <=) is true, then a match has been found.

Multiple checks can be made in each **case**, by separating them by a colon. If a match is found, the program finishes its checks, and goes on to execute the **case** block. No further expressions are evaluated or checked.

example of expression lists:

Case 1	constant
Case 5.4 To 10.1	range
Case Is > 3	bigger than-smaller than
Case 1, 3, 5, 7 to 9	match against a set of values
Case x	value of a variable

If **As Const** is used, only integer constants (all numeric constants except floating point constants: single and double) can be evaluated and the expressions are limited to constants and enumerations only. "To" ranges are supported, but "Is" is not.

With **As Const**, a jump table is created to contain the full range of integers. This allows **Select Case As Const** to be faster than **Select Case**. However, the range of values is limited, and the largest value in the range may be no higher than 8191.

Example

```
Dim choice As Integer
```

```

Input "Choose a number between 1 and 10: "; choice

Select Case As Const choice
Case 1
    Print "number is 1"
Case 2
    Print "number is 2"
Case 3, 4
    Print "number is 3 or 4"
Case 5 To 10
    Print "number is in the range of 5 to 10"
Case Else
    Print "number is outside the 1-10 range"
End Select

```

```

'' SELECT CASE vs. SELECT CASE AS CONST speed test

Const N = 50000000

Dim As Integer dummy = 0
Dim As Double t = Timer()

For i As Integer = 1 To N
    Select Case i
    Case 1, 3, 5, 7, 9
        dummy += 1
    Case 2, 4, 6, 8, 10
        dummy += 1
    Case 11 To 20
        dummy += 1
    Case 21 To 30
        dummy += 1
    Case 31
        dummy += 1
    Case 32

```

```

        dummy += 1
    Case 33
        dummy += 1
    Case Is >= 34
        dummy += 1
    Case Else
        Print "can't happen"
    End Select
Next

Print Using "SELECT CASE: ##.### seconds"; Timer()
t = Timer()

For i As Integer = 1 To N
    Select Case As Const i
        Case 1, 3, 5, 7, 9
            dummy += 1
        Case 2, 4, 6, 8, 10
            dummy += 1
        Case 11 To 20
            dummy += 1
        Case 21 To 30
            dummy += 1
        Case 31
            dummy += 1
        Case 32
            dummy += 1
        Case 33
            dummy += 1
        Case Else
            If( i >= 34 ) Then
                dummy += 1
            Else
                Print "can't happen"
            End If
        End Select
    End Select
Next

Print Using "SELECT CASE AS CONST: ##.### seconds"

```

Differences from QB

- **Select Case As Const** did not exist in QB
- in an "*expr1* TO *expr2*" case, QB would always evaluate both e was higher than the original expression.

See also

- **If...Then**

SetDate



Sets the current system date

Syntax

```
Declare Function SetDate ( ByRef newdate As Const String ) As Long
```

Usage

```
result = SetDate( newdate )
```

Parameters

newdate
the new date to set

Return Value

Returns zero on success or non-zero on failure on all ports except DOS.

Description

To set the date you just format *newdate* and send to **SetDate** in a valid format following one of the following: "mm-dd-yy", "mm-dd-yyyy", "mm/dd/yy", or "mm/dd/yyyy" (mm is the month, dd is the day, yy or yyyy the year).

Example

```
Dim m As String, d As String, y As String
m = "03" 'march
d = "13" 'the 13th
y = "1994" 'good ol' days
SetDate m + "/" + d + "/" + y
```

Differences from QB

- The DATE statement was used in QB and the syntax was "*DATE = string*"

See also

- [Date](#)
- [SetTime](#)

SetMouse



Sets the position and visibility of the mouse cursor

Syntax

```
Declare Function SetMouse ( ByVal x As Long = -1, ByVal y As Long = -1, ByVal clip As Long = -1 ) As Long
```

Usage

```
result = SetMouse([ x ] [, [ y ] [, [ visibility ] [, [ clip ]])
```

Parameters

(For each parameter, -1 is a special value indicating "no changes.")

x

optional - set x coordinate

y

optional - set y coordinate

visibility

optional - set visibility: 1 indicates visible, 0 indicates hidden

clip

optional - set clipping: 1 indicates mouse is clipped to graphics window

Return Value

Zero (0) on success, non-zero to indicate failure.

Description

`SetMouse` will set the (*x*, *y*) coordinates of the mouse pointer, as well as position is set using the *x* and *y* parameters. The mouse will be visible invisible if *visibility* is set to 0. `SetMouse` is intended for graphics mode (**Graphics**) statement only.

Example

```
Dim As Integer x, y, buttons
```

```

' create a screen 640*480
ScreenRes 640, 480
Print "Click the mouse button to center the mouse"

Do
    ' get mouse x, y and button state (wait until
Do: Sleep 1: Loop While GetMouse( x, y , , but

    If buttons And 1 Then
        ' on left mouse click, center mouse
        SetMouse 320, 240
    End If

    ' run loop until a key is pressed or the window is closed
Loop While Inkey = ""

```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- [GetMouse](#)
- [Screen](#)
- [MultiKey](#)
- [GetKey](#)

SetTime



Sets the current system time

Syntax

```
Declare Function SetTime ( ByRef newtime As Const String ) As Long
```

Usage

```
result = SetTime( newtime )
```

Parameters

newtime
the new time to set

Return Value

Returns zero on success or non-zero on failure on all ports except DOS.

Description

To set the time, format the date and send to `settime` in one of the following formats: "hh:mm:ss", "hh:mm", or "hh" (hh is the hour, mm is the minute, and ss is the second).

Example

```
SetTime "1:20:30"
```

Differences from QB

- The `Time` statement was used QB and the syntax was `TIME = newtime`.

See also

- [Time](#)
- [SetDate](#)

Returns the sign part of a number

Syntax

```
Declare Function Sgn ( ByVal number As Integer ) As Integer  
Declare Function Sgn ( ByVal number As LongInt ) As LongInt  
Declare Function Sgn ( ByVal number As Double ) As Double
```

Usage

```
result = Sgn( number )
```

Parameters

number
the number to find the sign of

Return Value

Returns the sign part of *number*.

- If *number* is greater than zero, then **sgn** returns 1.
- If *number* is equal to zero, then **sgn** returns 0.
- If *number* is less than zero, then **sgn** returns -1.

Description

The required *number* argument can be any valid numeric expression. Unsigned numbers will be treated as if they were signed, i.e. if the highest bit is set the number will be treated as negative, and -1 will be returned.

The **sgn** unary **operator** can be overloaded with user defined types.

Example

```
Dim N As Integer = 0
```

```
Print Sgn ( -1.87 )  
Print Sgn ( 0 )  
Print Sgn ( 42.658 )  
Print Sgn ( N )
```

The output would look like:

```
-1  
0  
1  
0
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- None

See also

- [Abs](#)
- [Operator](#)

Variable declaration modifier specifying visibility throughout a module

Syntax

```
Dim Shared ...  
ReDim Shared ...  
Common Shared ...  
Static Shared ...
```

Description

`shared` makes module-level variables visible inside `Subs` and `Function`. If `shared` is not used on a module-level variable's declaration, the variable is only visible in the level code in that file (furthermore, only a variable declared with `Dim W` inside a Namespace block, is stored on the stack).

NOTES (for `shared` variables excluding `Common` variables):

- Generally a `shared` variable may only be initialized with a constant value. The value is set at the start of the program in the `.data` section, so it cannot depend on any variables or functions in it).
- A first exception is a `shared` variable of var-len string type, even with a constant string (because of its structure with `VarLen` but to point to a dynamic memory block).
- A second exception is a `shared` variable of user-defined type, that can be initialized with a non-constant value (e.g. a code, called when the program starts, which writes the `data` section).

Example

```
' ' Compile with -lang qb or fblite  
  
'$lang: "qb"  
  
Declare Sub MySub  
Dim Shared x As Integer
```

```
Dim y As Integer

x = 10
y = 5

MySub

Sub MySub
    Print "x is "; x 'this will report 10 as it is
    Print "y is "; y 'this will not report 5 becau
End Sub
```

Differences from QB

- The **shared** statement inside scope blocks -- functions, subs, if, supported. Use `Dim|Redim|Common|Static shared` in the main pro a scope block and `Redim`ing a variable or array previously set without **shared**; it will work fine and won't ruin anything.

See also

- **Common**
- **Dim**
- **Erase**
- **Extern**
- **LBound**
- **ReDim**
- **Preserve**
- **Static**
- **UBound**
- **Var**

Shell



Sends a command to the system command interpreter

Syntax

```
Declare Function Shell ( ByRef command As Const String ) As Long
```

Usage

```
result = Shell( command )
```

Parameters

command

A string specifying the command to send to the command interpreter.

Return Value

If the command could not be executed, -1 is returned. Otherwise, the command is executed and its exit code is returned.

Description

Program execution will be suspended until the command interpreter exits.

Example

```
'e.g. for windows:  
Shell "dir c:*.*)"

'e.g. for linux:  
Shell "ls"
```

Platform Differences

- Linux requires the *command* case matches the real name of th

command. Windows and DOS are case insensitive. The program being shelled may be case sensitive for its command line parameters.

- Path separators in Linux are forward slashes / . Windows uses backward slashes \ but it allows for forward slashes. DOS uses backward \ slashes.
- If an empty *command* string is passed, DOS will open an interactive command prompt. On Windows, an error may be returned.

Differences from QB

- QB allowed SHELL on its own without a "command" argument which caused a default command shell to be started. Execution in the main program would suspend until exit from the command shell. The behaviour in FB is platform-dependent.

See also

- [Exec](#)
- [Run](#)

Operator Shl (Shift Left)



Shifts the bits of a numeric expression to the left

Syntax

```
Declare Operator Shl ( ByRef lhs As Integer, ByRef rhs As Integer ) As Integer
Declare Operator Shl ( ByRef lhs As UInteger, ByRef rhs As UInteger ) As UInteger
Declare Operator Shl ( ByRef lhs As LongInt, ByRef rhs As LongInt ) As LongInt
Declare Operator Shl ( ByRef lhs As ULongInt, ByRef rhs As ULongInt ) As ULongInt
```

Usage

```
result = lhs Shl rhs
```

Parameters

lhs

The left-hand side expression.

rhs

The right-hand side shift expression.

Return Value

Returns the result of *lhs* being shifted left *rhs* number of times.

Description

operator shl (Shift left) shifts all of the bits in the left-hand side expression (*lhs*) left a number of times specified by the right-hand side expression (*rhs*). Numerically, the result is the same as "**CInt**(*lhs* * \wedge *rhs*)". For example, "&b0101; **shl** 1" returns the binary number &b01010; and "5 **shl** 1" returns 10.

Neither of the operands are modified in any way.

If the result is too large to fit inside the result's data type, the leftmost bits are discarded ("shifted out").

The results of this operation are undefined for values of *rhs* less than zero, or greater than or equal to the number of bits in the result's data type.

This operator can be overloaded for user-defined types.

Example

```
'Double a number
For i As Integer = 0 To 10

    Print 5 Shl i, Bin(5 Shl i, 16)

Next i
```

Output:

5	0000000000000101
10	0000000000001010
20	0000000000010100
40	0000000000101000
80	0000000001010000
160	0000000010100000
320	0000000101000000
640	0000001010000000
1280	0000010100000000
2560	0000101000000000
5120	0001010000000000

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__sh1`.

Differences from QB

- New to FreeBASIC

See also

- **Operator Shl= (Shift Left And Assign)**
- **Operator Shr (Shift Right)**
- **Bin**
- **Mathematical Functions**

Operator Shr (Shift Right)



Shifts the bits of a numeric expression to the right

Syntax

```
Declare Operator Shr ( ByRef lhs As Integer, ByRef rhs As Integer ) As Integer
Declare Operator Shr ( ByRef lhs As UInteger, ByRef rhs As UInteger ) As UInteger
Declare Operator Shr ( ByRef lhs As LongInt, ByRef rhs As LongInt ) As LongInt
Declare Operator Shr ( ByRef lhs As ULongInt, ByRef rhs As ULongInt ) As ULongInt
```

Usage

```
result = lhs Shr rhs
```

Parameters

lhs

The left-hand side expression.

rhs

The right-hand side shift expression.

Return Value

Returns the result of *lhs* being shifted right *rhs* number of times.

Description

operator shr (Shift right) shifts all of the bits in the left-hand side expression (*lhs*) right a number of times specified by the right-hand side expression (*rhs*). Numerically, the result is the same as "**Int**(*lhs* 2^{-rhs})". For example, "&b0101; shr 1" returns the binary number &b010;, and "5 shr 1" returns 2.

If the left-hand side expression is signed and negative, the sign bit is copied in the newly created bits on the left after the shift. For example "-5 shr 2" returns -2.

Neither of the operands are modified in any way.

The results of this operation are undefined for values of *rhs* less than zero, or greater than or equal to the number of bits in the result's data type.

This operator can be overloaded for user-defined types.

Example

```
'Halve a number
For i As Integer = 0 To 10

    Print 1000 Shr i, Bin(1000 Shr i, 16)

Next i
```

Output:

```
1000      0000001111101000
500       0000000111110100
250       0000000011111010
125       0000000001111101
62        0000000000111110
31        0000000000011111
15        0000000000001111
7         0000000000000111
3         0000000000000011
1         0000000000000001
0         0000000000000000
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__shr`.

Differences from QB

- New to FreeBASIC

See also

- **Operator Shr= (Shift Right And Assign)**
- **Operator Shl (Shift Left)**
- **Bin**
- **Mathematical Functions**

Short



Standard data type: 16 bit signed

Syntax

```
Dim variable As Short
```

Description

16-bit signed whole-number data type. Can hold values from -32768 to 32767.

Example

```
Dim x As Short = CShort(&H8000)
Dim y As Short = CShort(&H7FFF)
Print "Short Range = "; x; " to "; y
```

Output:

```
Short Range = -32768 to 32767
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__Short`.

Differences from QB

- The name "short" is new to FreeBASIC, however they are the same as integers in QB

See also

- UShort
- CShort

Returns the sine of an angle

Syntax

```
Declare Function Sin ( ByVal angle As Double ) As Double
```

Usage

```
result = Sin( angle )
```

Parameters

angle
the angle (in radians)

Return Value

Returns the sine of the argument *angle* as a **Double** within the range o

Description

The argument *angle* is measured in **radians** (not **degrees**).

The value returned by this function is undefined for values of *angle* wi
value of 2^{63} or greater.

Example

```
Const PI As Double = 3.1415926535897932
Dim a As Double
Dim r As Double
Input "Please enter an angle in degrees: ", a
r = a * PI / 180      'Convert the degrees to Radian
Print ""
Print "The sine of a" ; a; " degree angle is"; Sin(r)
Sleep
```

The output would look like:

```
Please enter an angle in degrees: 30
The sine of a 30 degree angle Is 0.5
```

Differences from QB

- None

See also

- [Asin](#)
- [Cos](#)
- [Tan](#)
- [A Brief Introduction To Trigonometry](#)

Single



Standard data type: 32 bit floating point

Syntax

```
Dim variable As Single
```

Description

Single is a 32-bit, floating point data type used to store decimal numbers. They can hold positive values in the range $1.401298e-45$ to $3.402823e+38$, or negative values in the range $-1.401298e-45$ to $-3.402823e+38$, or zero (0). They contain at most 24 bits of precision, or about 6 decimal digits.

They are similar to **Double** data types, but less precise.

Example

```
'Example of using a single variable.  
  
Dim a As Single  
a = 1.9857665  
Print a  
  
Sleep
```

Differences from QB

- None

See also

- **Double** More precise float type
- **CSng**

- Table with variable types overview, limits and suffixes

Returns the size of a variable or type in bytes.

Syntax

```
sizeof ( variable | DataType )
```

Description

The **sizeof** operator returns the number of bytes taken up by a *variable* Or **DataType**.

Different from **Len**, when used with fixed-length strings (including fixed length **zStrings** and **wStrings**) it will return the number of bytes they use, and when used with variable-length strings, it will return the size of the string descriptor.

If there is both a user defined type and a variable visible with the same name in the current scope, the user defined type takes precedence over the variable. To ensure that the **sizeof** takes the variable instead of the user defined type, wrap the argument to **sizeof** with parentheses to force it to be seen as an expression. For example `sizeof((variable))`.

Note: When used with arrays, **sizeof** returns the size of a single element of the array. This differs from its behavior in C, where arrays could only be a fixed size, and `sizeof()` would return the number of it used.

For clarity, it is recommended that you avoid this potential confusion, and use **sizeof** directly on an array element, rather than the whole array.

Remark: When used with a dereferenced z/wstring pointer, **sizeof** always returns the number of bytes taken up by one z/wstring character (instead of 0 before fbc version 0.90).

Example

```
Print SizeOf(Byte) ' returns 1
```

```
Type bar  
  a As Integer  
  b As Double  
End Type  
Dim foo As bar  
Print SizeOf(foo)
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__sizeof`.

Differences from QB

- New to FreeBASIC

See also

- [Len](#)

Sleep



Waits until a specified time has elapsed, or a key is pressed.

Syntax

```
Declare Sub Sleep ( ByVal amount As Long = -1 )
Declare Function Sleep ( ByVal amount As Long , ByVal keyflag As
Long ) As Long
```

Usage

```
Sleep [ amount [, keyflag ]]
result = Sleep ( amount, keyflag )
```

Parameters

amount

Optional number of milliseconds to wait (default is to wait for a key press).

keyflag

Optional flag; give it a value of 0 for a normal sleep, or 1 to specify that the wait cannot be interrupted by a key press.

Return Value

Returns 1 if *keyflag* was not a valid value (i.e. something other than 0 or 1) to indicate failure, or 0 otherwise.

Description

`sleep` will wait until *amount* milliseconds (can be seconds in *-lang qb*, see below) given elapsed (if any value was passed) or until the user presses a key. If *amount* is below 100 ms then `sleep` will always wait the full requested amount (key presses are ignored).

Include the second parameter, 1, for a "deep" sleep, which cannot be interrupted by pressing a key.

The accuracy of `sleep` is variable depending on the OS cycle time (Windows NT/2K/XP: 15 ms, 9x/Me: 50 ms, Linux 10ms, DOS 55 ms)

Call `sleep` with 25ms or less to release time-slice when waiting for user input or looping inside a thread. This will prevent the program from unnecessarily hogging the CPU.

`sleep` does not clear the keyboard buffer and any keys pressed during a call to `sleep` are retained and can be read using `Inkey`. In order to wait for a key press, and remove the key from the buffer, `getKey` can be used instead.

Example

```
Print "press a key"
Sleep
GetKey 'clear the keyboard buffer
Print "waiting half second"
Sleep 500
```

Dialect Differences

- In the *-lang fb* and *-lang fblite* dialects, the *amount* value is in **milliseconds**.
- In the *-lang qb* dialect, the *amount* value is in **seconds** as in QB. If the second parameter *keyflag* is given, or the keyword is written as `__sleep` the value is expected to be in **milliseconds**.

Differences from QB

- None in the *-lang qb* dialect.
- In QB, the delay was given in whole seconds only and did not support the *keyflag* parameter.

See also

- [Timer](#)

- Inkey

Space



Creates a string of a given length filled with spaces (" ")

Syntax

```
Declare Function Space( ByVal count As Integer ) As String
```

Usage

```
result = Space[$]( count )
```

Parameters

count

An integer type specifying the length of the string to be created.

Return Value

The created string. An empty string will be returned if *count* <= 0.

Description

`space` creates a string with the specified number of spaces.

Example

```
Dim a As String
a = "x" + Space(3) + "x"
Print a ' prints: x   x
```

Dialect Differences

- The string type suffix \$ is obligatory in the *-lang qb* dialect.
- The string type suffix \$ is optional in the *-lang fblite* and *-lang fb* dialects.

Differences from QB

- None

See also

- WSpace
- Spc
- String (Function)

Spc



Output function to skip spaces when writing to screen or file

Syntax

```
Spc( columns )
```

Usage

```
Print Spc( spaces ) [(, | ;)] ...
```

Parameters

spaces
number of spaces to skip

Description

`spc` skips over the given number of *spaces* when **Printing** to screen or to a file. The character cells skipped over are left unchanged.

Example

```
Print "foo"; Spc(5); "bar"  
Print "hello"; Spc(4); "world"
```

```
' ' Uses Spc to justify text instead of Tab  
  
Dim As String A1, B1, A2, B2  
  
A1 = "Jane"  
B1 = "Doe"  
A2 = "Bob"  
B2 = "Smith"  
  
Print "FIRST NAME"; Spc(35 - 10); "LAST NAME"
```

```
Print "-----"; Spc(35 - 10); "-----"  
Print A1; Spc(35 - Len(A1)); B1  
Print A2; Spc(35 - Len(A2)); B2
```

The output would look like:

FIRST NAME -----	LAST NAME -----
Jane	Doe
Bob	Smith

Differences from QB

- In QBASIC, spaces were printed in the gap, while in FreeBASIC, the characters are just skipped over and left untouched. The `Space` function can still be used to achieve this effect.

See also

- `Tab`
- `Space`
- `(Print | ?)`

Returns a square root of a number

Syntax

```
Declare Function Sqr ( ByVal number As Double ) As Double
```

Usage

```
result = Sqr( number )
```

Parameters

number

the number (greater than or equal to zero)

Return Value

Returns the square root of the argument *number*.

If *number* equals zero, **sqr** returns zero (0.0).

If *number* is less than zero, **sqr** returns a special value representing "NaN" or "IND", exact text is platform dependent.

Description

This is the same as raising the argument *number* to the one-half power required *number* argument can be any valid numeric expression greater

If a **LongInt** or **ULongInt** is passed to **sqr**, it may be converted to **Double** numbers over 2^{52} , this will cause a very small loss of precision. With assumptions about the rounding method, the maximum error due to the **sqr**($2^{64}-2^{12}$), which is about $4.8e-7$. However this may cause error ceiling of this value is taken, and the result of this may be out by 1, particularly for large numbers and numbers that are close by.

Example

```
' ' Example of Sqr function: Pythagorean theorem
Dim As Single a, b

Print "Pythagorean theorem, right-angled triangle"
Print
Input "Please enter one leg side length: ", a
Input "Please enter the other leg side length: ",
Print
Print "The hypotenuse has a length of: " & Sqr( a
```

The output would look like:

```
Pythagorean theorem, right-angled triangle

Please enter one leg side length: 1.5
Please enter the other leg side length: 2

The hypotenuse has a length of: 2.5
```

Differences from QB

- None

See also

- **Operator ^ (Exponentiate)**
- **Arithmetic Operators**

Defines variables, objects and arrays having static storage

Syntax

```
Static symbol1 [ (array-dimensions) ] As DataType [ = expression  
[ (array-dimensions) ] As DataType [ = expression], ...]
```

or

```
Static As DataType symbol1 [ (array-dimensions) ] [ = expression  
[ (array-dimensions) ] [ = expression], ...]
```

or

```
Sub|Function procedurename ( parameters ) [As DataType] Static  
...  
End Sub|Function
```

Parameters

symbol

variable or array symbol name.

array-dimensions

lower-bound To upper-bound [, ...]

or

Any [, **Any**...]

or *empty*.

expression

An constant expression, or an array of constant expressions

Description

Specifies **static storage** for variables, objects and arrays; they are all program startup and deallocated upon exit. Objects are constructed only if they are defined, and destructed upon program exit.

When declaring static arrays, only **numeric literals**, **Constants** or **Empty** may be used as subscript range values. Static variable-length arrays may be declared empty (no subscript range list) and resized using **ReDim** before use.

In both iterative and recursive blocks, like looping **control flow statements** and procedures, static variables, objects and arrays local to the block are not destroyed when the block ends.

occupy the same storage across all instantiations of the block. For example, recursive procedures that call themselves - either directly or indirectly - share the instances of their local static variables.

A static variable may only be initialised with a constant value: its start is at the start of the program before any code is run, and so it cannot depend on variables or functions in it.

When used with module-level and member procedure declarations, **static storage** for all local variables, objects and arrays.

At module-level variable declaration only, the modifier **shared** may be used instead of the keyword **static** to make module-level static variables visible inside procedures.

When used with in a user-defined type, **static** creates **Static Members Or Variables**.

Example

```
Sub f
  ' times called is initially 0
  Static timesCalled As Integer = 0
  timesCalled += 1
  Print "Number of times called: " & timesCalled
End Sub

' the static variable in f() retains its value between
' multiple procedure calls.
f()
f()
```

Will output:

```
Number of times called: 1
Number of times called: 2
```

Dialect Differences

- Variables cannot be initialised in the *-lang qb* dialect.

Differences from QB

- QuickBASIC allows variables and arrays to be declared using the `Dim` keyword within procedures and `DEF FN` routines only.
- `static` forces local visibility of variables and arrays in QuickBASIC routines. FreeBASIC supports neither `DEF FN` routines nor this `Static`.

See also

- `Static (Member)`
- `Dim, ReDim`
- `Shared`
- `Sub (Module), Function (Module)`
- `Sub (Member), Function (Member)`
- `Option Static`
- `Storage Classes`

Static (Member)



Declare a static member procedure or variable

Syntax

```
Type typename
Static variablename As DataType [, ...]
Declare Static Sub|Function procedurename ...
...
End Type

Dim typename.variablename As DataType [= initializer] [, ...]

[Static] Sub|Function typename.procedurename ...
...
End Sub|Function
```

Description

- Static member procedures

static methods do not have an implicit **This** instance argument passed to member procedures (for example with callback procedure pointers). A **static** method is encapsulated in the *typename* namespace, and therefore have the ability to be called on instances of *typename*.

static methods can be called directly anywhere in code, like normal methods. Similar to non-static methods, however either way there is no implicit **This** argument for a **static** method.

For member procedures with a **static** declaration, **static** may also be used to improve code readability.

- Static member variables

static member variables are created and initialized only once independent of ("instance") member variables which are created again and again for each instance, even if **shared** was not specified in the declaration. Thus, **static** member variables are declared in a **Type** namespace.

Each **static** member variable declared in a **Type** must be explicitly declared in a **Static** statement. The declaration inside the **Type** is the prototype that is visible to all instances.

definition outside the Type allocates and optionally initializes the `static` member variable: it can only be allocated in a single module, not in multiple modules.

A `static` member variable is subject to member access control except that a `static` member variable is to be explicitly initialized outside the Type's member definition.

Example

```
' ' Example showing how the actual procedure invoked
' ' using static member procedures.
Type _Object

    Enum handlertype
        ht_default
        ht_A
        ht_B
    End Enum

    Declare Constructor( ByVal ht As handlertype = ht_default )
    Declare Sub handler()

Private:
    Declare Static Sub handler_default( ByRef obj As _Object )
    Declare Static Sub handler_A( ByRef obj As _Object )
    Declare Static Sub handler_B( ByRef obj As _Object )
    Declare Static handler_func As Sub( ByRef obj As _Object )

End Type

Constructor _Object( ByVal ht As handlertype )
    Select Case ht
    Case ht_A
        handler_func = @_Object.handler_A
    Case ht_B
        handler_func = @_Object.handler_B
    End Select
```

```

    Case Else
        handler_func = @_Object.handler_default
    End Select
End Constructor

Sub _Object.handler()
    handler_func(This)
End Sub

Sub _Object.handler_default( ByRef obj As _Object
    Print "Handling using default method"
End Sub

Sub _Object.handler_A( ByRef obj As _Object )
    Print "Handling using method A"
End Sub

Sub _Object.handler_B( ByRef obj As _Object )
    Print "Handling using method B"
End Sub

Dim objects(1 To 4) As _Object => _
    {
        _Object.handlertype.ht_B, _
        _Object.handlertype.ht_default, _
        _Object.handlertype.ht_A _
    }
    '' 4th array item will be _Object.handlertype.ht

For i As Integer = 1 To 4
    Print i,
    objects(i).handler()
Next i

```

```

'' Assign an unique ID to every instance of a Type

```

```

Type UDT
  Public:
    Declare Property getID () As Integer
    Declare Constructor ()
  Private:
    Dim As Integer ID
    Static As Integer countID
End Type
Dim As Integer UDT.countID = 0

Property UDT.getID () As Integer
  Property = This.ID
End Property

Constructor UDT ()
  This.ID = UDT.countID
  UDT.countID += 1
End Constructor

Dim As UDT uFirst
Dim As UDT uSecond
Dim As UDT uThird

Print uFirst.getID
Print uSecond.getID
Print uThird.getID

```

Differences from QB

- New to FreeBASIC

See also

- [Class](#)
- [Declare](#)

- **Type**
- **Static**

Specifies a *stdcall*-style calling convention in a procedure declaration

Syntax

```
Sub name stdcall [Overload] [Alias "alias"] ( parameters )  
Function name stdcall [Overload] [Alias "alias"] ( parameters )
```

Description

In procedure declarations, `stdcall` specifies that a procedure will use registers to be passed (pushed onto the stack) in the reverse order in which they are used. The caller must clean up the stack (pop any parameters from EAX, ECX or EDX registers, and must clean up the stack (pop any parameters).

`stdcall` is not allowed to be used with variadic procedure declarations.

`stdcall` is the default calling convention on Windows, unless another calling convention is specified. `stdcall` is also the standard (or most common) calling convention used in Windows API.

Example

```
Declare Function Example stdcall (param1 As Integer, param2 As Integer)  
Declare Function Example2 cdecl (param1 As Integer, param2 As Integer)  
  
Function Example stdcall (param1 As Integer, param2 As Integer)  
    ' This is an STDCALL function, the first parameter is passed in EAX  
    Print param1, param2  
    Return param1 Mod param2  
End Function  
  
Function Example2 cdecl (param1 As Integer, param2 As Integer)  
    ' This is a CDECL function, the first parameter is passed in ECX  
    Print param1, param2  
    Return param1 Mod param2  
End Function
```

Platform Differences

- On Windows systems, `stdcall` procedures have an "@N" decorator list, in bytes.

Differences from QB

- New to FreeBASIC

See also

- `pascal`, `cdecl`
- `Declare`
- `Sub`, `Function`

Step



Statement modifier.

Syntax

For *iterator* = *initial_value* To *end_value* **Step** *increment*

Line [*buffer*,] **Step** (*x1*, *y1*) - **Step** (*x2*, *y2*) [, [*color*] [*B|BF*] [, *style*]]]

Circle [*target*,] **Step** (*x*, *y*), *radius* [, [*color*] [, [*start* [, [*end*] [, [*aspect*] [, *F*]]]]]]

Paint [*target*,] **STEP** (*x*, *y*) [, [*paint*] [, [*border_color*]]]

Description

In a **For** statement, **step** specifies the increment of the loop iterator with each loop.

In a **Line**, **Circle** or **Paint** statement, **step** indicates that the following coordinate has values relative to the graphics cursor.

Example

```
Dim i As Integer
For I=10 To 1 Step -1
Next
```

```
Line -Step(10,10),13
```

See also

- **For...Next**
- **Line**
- **Circle**
- **Paint**

Reads axis position from attached gaming devices

Syntax

```
Declare Function Stick ( ByVal axis As Long ) As Long
```

Usage

```
result = Stick( axis )
```

Parameters

axis

the axis number to query for position

Return Value

Returns a number between 1 and 200 for specified *axis*, otherwise zero if the device is not attached.

Description

`stick` will retrieve the axis position for the first and second axes on the second gaming devices. *axis* must be a number between 0 and 3 having the following meaning:

Axis	Returns
0	X position of gaming device A
1	Y position of gaming device A when STICK(0) was called
2	X position of gaming device B when STICK(0) was called
3	Y position of gaming device B when STICK(0) was called

`stick(0)` must first be called to obtain the positions for the other axes.

Example

```

'' Compile with -lang qb

'$lang: "qb"

Screen 12

Do
    Locate 1, 1
    Print "Joystick A-X position : "; Stick(0); "
    Print "Joystick A-Y position : "; Stick(1); "
    Print "Joystick B-X position : "; Stick(2); "
    Print "Joystick B-Y position : "; Stick(3); "
    Print
    Print "Button A1 was pressed : "; Strig(0); "
    Print "Button A1 is pressed : "; Strig(1); "
    Print "Button B1 was pressed : "; Strig(2); "
    Print "Button B1 is pressed : "; Strig(3); "
    Print "Button A2 was pressed : "; Strig(4); "
    Print "Button A2 is pressed : "; Strig(5); "
    Print "Button B2 was pressed : "; Strig(6); "
    Print "Button B2 is pressed : "; Strig(7); "
    Print
    Print "Press ESC to Quit"

    If Inkey$ = Chr$(27) Then
        Exit Do
    End If

    Sleep 1

Loop

```

Dialect Differences

- Only available in the *-lang qb* dialect.

Differences from QB

- None

See also

- `GetJoystick`
- `Strig`

Stop



Halts program execution, and waits for a key press before ending the program.

Syntax

```
Declare Sub Stop ( ByVal retval As Long = 0 )
```

Usage

Stop

Parameters

retval

Error code returned to system.

Description

Halts the execution of the program and stands by. It's provided as a help to debugging, as it preserves the memory and doesn't close files. For normal program termination the **End** keyword should be used. An optional return value, an integer, can be specified to return an error code to the system. If no return value is given, a value of 0 is automatically returned.

Note: STOP is not implemented properly yet; currently it is the same as **System**.

Example

```
Print "this text is shown"  
Sleep  
Stop  
Print "this text will never be shown"
```

Differences from QB

- None

See also

- End

Returns a string representation of a number, boolean or Unicode character string

Syntax

```
Declare Function Str ( ByVal n As Byte ) As String
Declare Function Str ( ByVal n As UByte ) As String
Declare Function Str ( ByVal n As Short ) As String
Declare Function Str ( ByVal n As UShort ) As String
Declare Function Str ( ByVal n As Long ) As String
Declare Function Str ( ByVal n As ULong ) As String
Declare Function Str ( ByVal n As LongInt ) As String
Declare Function Str ( ByVal n As ULongInt ) As String
Declare Function Str ( ByVal n As Single ) As String
Declare Function Str ( ByVal n As Double ) As String
Declare Function Str ( ByVal b As Boolean ) As String
Declare Function Str ( ByRef str As Const String ) As String
Declare Function Str ( ByVal str As Const WString ) As String
```

Usage

```
result = Str[$]( number )
or
result = Str( string )
```

Parameters

number

Numeric expression to convert to a string.

string

String expression to convert to a string.

Description

`str` converts numeric variables to their string representation. Used this way it is the `string` equivalent to `wstr` applied to numeric variables, and the opposite of the `val` function, which converts a string into a number.

`str` converts boolean variables to their string representation "false" / "true".

`str` also converts Unicode character strings to ASCII character strings. Used this way it does the opposite of `wstr`. If an ASCII character string is given, that string is returned unmodified.

Example

```
Dim a As Integer
Dim b As String
a = 8421
b = Str(a)
Print a, b
```

Dialect Differences

- In the *-lang qb* dialect, `str` will left pad a positive number with space.
- The string type suffix "\$" is obligatory in the *-lang qb* dialect.
- The string type suffix "\$" is optional in the *-lang fblite* and *-lang fb* dialects.

Platform Differences

- DOS version/target of FreeBASIC does not support the wide-character string version of `str`.

Differences from QB

- QB does not support the wide-character string version of `str`.

See also

- `Val`
- `Cbool`
- `Chr`

■ Asc

Reads button state from attached gaming devices

Syntax

Declare Function Strig (ByVal *button* As Long) As Long

Usage

result = **Strig**(*button*)

Parameters

button

the button to query for state

Return Value

Returns -1 (pressed) or 0 (not-pressed) to indicate the state of the *but* requested.

Description

strig will retrieve the button state for the first and second buttons on 1 and second gaming devices. *button* must be a number between 0 and 7 and has the following meaning:

Button	State to return
0	First button on gaming device A pressed since STICK(0) was called
1	First button on gaming device A is pressed
2	First button on gaming device B pressed since STICK(0) was called
3	First button on gaming device B is pressed
4	Second button on gaming device A pressed since STICK(0) was called
5	Second button on gaming device A is pressed
6	Second button on gaming device B pressed since STICK(0) was called
7	Second button on gaming device B is pressed

Calling `stick(0)` will reset the state returned where *button* is equal to 6.

Example

```
' ' Compile with -lang qb
'$lang: "qb"

Screen 12

Do
  Locate 1, 1
  Print "Joystick A-X position : "; Stick(0); "
  Print "Joystick A-Y position : "; Stick(1); "
  Print "Joystick B-X position : "; Stick(2); "
  Print "Joystick B-Y position : "; Stick(3); "
  Print
  Print "Button A1 was pressed : "; Strig(0); "
  Print "Button A1 is pressed : "; Strig(1); "
  Print "Button B1 was pressed : "; Strig(2); "
  Print "Button B1 is pressed : "; Strig(3); "
  Print "Button A2 was pressed : "; Strig(4); "
  Print "Button A2 is pressed : "; Strig(5); "
  Print "Button B2 was pressed : "; Strig(6); "
  Print "Button B2 is pressed : "; Strig(7); "
  Print
  Print "Press ESC to Quit"

  If Inkey$ = Chr$(27) Then
    Exit Do
  End If

  Sleep 1

Loop
```

Dialect Differences

- Only available in the *-lang qb* dialect.

Differences from QB

- None

See also

- [GetJoystick](#)
- [Stick](#)

String (Function)



Creates and fills a string of a certain length with a certain character

Syntax

```
Declare Function String ( ByVal count As Integer, ByVal ch_code  
As Long ) As String  
Declare Function String ( ByVal count As Integer, ByRef ch As  
Const String ) As String
```

Usage

```
result = String[$]( count, ch_code )  
or  
result = String[$]( count, ch )
```

Parameters

count

An integer specifying the length of the string to be created.

ch_code

A long specifying the ASCII character code to be used to fill the string.

ch

A string whose first character is to be used to fill the string.

Return Value

The created string. An empty string will be returned if either *ch* is an empty string, or *count* <= 0.

Description

A list of **ASCII character codes**.

Example

```
Print String( 4, 69 )           ' ' prints "EEEE"  
Print String( 5, "Indeed" )    ' ' prints "IIIII"  
End 0
```

Dialect Differences

- The string type suffix "\$" is obligatory in the *-lang qb* dialect.
- The string type suffix "\$" is optional in the *-lang fblite* and *-lang fb* dialects.

Differences from QB

- None

See also

- **String** (data type)
- **Space**

Standard data type: 8 bit character string

Syntax

```
Dim variable As String [ * size]
```

Description

A `string` is an array of characters.

A `string` declared without the `size` parameter is dynamically resized c bytes to 2 gigabytes. A descriptor contains a pointer to the actual string. `VarPtr` will return a pointer to the descriptor, while `StrPtr` will point to the string data.

Because of the hidden descriptor with a `string`, manual allocation of s (preferentially), for a `string` is not encouraged. The common way to e unnecessary allocations inside a loop for instance, is to use the `Space` function.

Nevertheless if necessary, dynamic allocation may be carefully used b `Reallocate` (see precautions for use) and string pointer (which is a poi to hold string descriptors, the string must always be destroyed (setting the memory taken up by the string data), otherwise, it is not possible t continuation.

Despite the use of the descriptor, an implicit NULL character (`Chr(0)`) is external libraries without making slow copies. FreeBASIC's internal fu

A `string` declared with a *fixed size* is a QB-style fixed length string, v what "-lang" compiler option is used. It has no descriptor and it is not i string, it is truncated on the right side.

Fixed length strings are also terminated with a NULL character, and so removed in future, to prevent the redundant character complicating da

String variable names need not end in a dollar sign \$ as in other diale disallowed entirely.

Example

```
' ' Variable length
Dim a As String

a = "Hello"
Print a

a += ", world!"
Print a

Var b = "Welcome to FreeBASIC"
Print b + "! " + a
```

```
' ' QB-like $ suffixes
#lang "qb"

' ' DIM based on $ suffix
Dim a$
a$ = "Hello"

' ' Implicit declaration based on $ suffix
b$ = ", world!"

Print a$ + b$
```

```
' ' Variable-length strings as buffers

' ' Reserving space for a string,
' ' using Space() to produce lots of space characters
Var mybigstring = Space(1024)
Print "buffer address: &h" & Hex( StrPtr( mybigstr
```

```
' ' Explicitly destroying a string
mybigstring = ""
Print "buffer address: &h" & Hex( StrPtr( mybigstr
```

```
' ' Variable-length string as Const parameter

' ' Const qualifier preventing string from being modified
Sub silly_print( ByRef printme As Const String )
    Print ".o0( " & printme & " )0o."
    'next line will cause error if uncommented
    'printme = "silly printed"
End Sub

Var status = "OK"

silly_print( "Hello FreeBASIC!" )
silly_print( "Status: " + status )
```

Differences from QB

- In QB the strings were limited to 32767 characters.
- In QB, the unused characters of a fixed-length string were initialized to zero.
- In QB static or fixed-size strings were often used in records to represent 1 byte in a UDT read from a file. This is not possible in VB6. The following. When converting QBASIC code that reads UDTs from a file, you must use `uByte (0 to n - 1)` or your files will be incompatible.

See also

- [String \(Function\)](#)
- [Space](#)
- [ZString](#)

- `WString`
- `Str`
- `StrPtr`
- `VarPtr`

Operator StrPtr (String Pointer)



Returns the address of a string's character data.

Syntax

```
Declare Operator StrPtr ( ByRef lhs As String ) As ZString Ptr  
Declare Operator StrPtr ( ByRef lhs As WString ) As ZString Ptr
```

Usage

```
result = StrPtr ( lhs )
```

Parameters

lhs
A string.

Return Value

Returns a **ZString Ptr** to a string's character data.

Description

This operator returns a **ZString Ptr** that points to the beginning of a string. **StrPtr** is the proper method for acquiring the address of a string's character data.

Note that when passed a **WString**, **Operator StrPtr** still returns a **ZString Ptr** as the desired result.

The related **Operator VarPtr (Variable Pointer)** and **Operator @ (Address of)** return the address of the internal string descriptor.

Example

```
' ' This example uses StrPtr to demonstrate using pointers  
Dim myString As String  
Dim toMyStringDesc As Any Ptr  
Dim toMyString As ZString Ptr
```

```

'' Note that using standard VARPTR notation will r
'' descriptor, not the string data itself
myString = "Improper method for Strings"
toMyStringDesc = @myString
Print myString
Print Hex( toMyStringDesc )
Print

'' However, using StrPtr returns the proper pointe
myString = "Hello World Examples Are Silly"
toMyString = StrPtr(myString)
Print myString
Print *toMyString
Print

'' And the pointer acts like pointers to other typ
myString = "MyString has now changed"
Print myString
Print *toMyString
Print

```

Differences from QB

- New to FreeBASIC, but does exactly the same thing as **SAdd**

See also

- **SAdd**
- **VarPtr**
- **ProcPtr**
- **Pointers**

Sub



Defines a procedure

Syntax

```
[Public|Private] Sub identifier [cdecl|pascal|stdcall] [Overload]
[( [parameter_list] )] [Static] [Export]
statements
```

```
...
[Return]
```

```
...
End Sub
```

```
[Public] Sub identifier [cdecl|pascal|stdcall] [Overload] [Alias]
[Constructor|Destructor] [Static]
statements
```

```
...
[Return]
```

```
...
End Sub
```

Parameters

identifier: the name of the subroutine

external_identifier: externally visible (to the linker) name enclosed in quotes

parameter_list: parameter[, parameter[, ...]]

parameter: [ByRef|ByVal] *identifier* [As *type*] [= *default_value*]

identifier: the name of the variable referenced in the subroutine

type: the type of variable

default_value: the value of the argument if none is specified in the call

statements: one or more statements that make up the subroutine body

Description

A subroutine is a block of code which may be called at any time from the main program. It can be executed multiple times, and subroutines provide an invaluable way of organizing code by replacing these blocks of code with a single subroutine call. A subroutine is a way to extend the FreeBASIC language to provide custom commands. Many of the standard FreeBASIC functions are merely subroutines part of a "runtime library" linked to the executable.

The **sub** keyword marks the beginning of a subroutine, and its end is marked by the **end sub** keyword.

parameter is the name by which this subroutine is called. For instance `Sub foo`, the user can execute the code in between `Sub foo` and `End Sub`. This code is executed separate from the code which calls the subroutine unless they are shared, are not available to the subroutine. Values can be passed to parameters.

Parameters are the arguments passed to any statement. For instance as `Print 4`, the value "4" is passed to the function `Print`. Parameters of a subroutine are supplied by one or more parameter arguments in the `Sub` statement. A subroutine with `Sub mysub(foo, bar) . . . End Sub`, allows the code in the subroutine to refer to the first passed argument as "foo" and the second passed argument as "bar". If a parameter is given a default value, that parameter is optional.

In the default dialect *-lang fb*, parameters must also have a supplied type. Type suffixes are not allowed.

In the *-lang qb* and *-lang fblite* dialects only, it will be given a default type either by name or by type suffix. The default type is `Single` in the *-lang fblite* dialect.

A subroutine can also specify how parameters are passed, either as `ByRef` or `ByVal` in the syntax definition. If a parameter is `ByRef`, the parameter name literally refers to the original variable passed to the subroutine. Any changes made to that variable will affect the original variable. If a parameter is passed `ByVal`, however, the value is passed into a new variable, and any changes made to it will not affect the original variable. `ByVal` is currently apply to `Strings`, and `ByVal` should be avoided with them for `Strings`.

The `Static` specifier indicates that the values of all local variables defined in the subroutine are preserved between calls. To specify individual local variables as static, use `Static` before the variable name.

`Sub` is the same as `Function`, except it does not allow a value to be returned.

The second syntax defines either a constructor or destructor using the `Constructor` or `Destructor` keywords, respectively. Constructor subroutines are executed before the module is loaded, while destructors execute on module exit. Note the public access list for both constructors and destructors.

Example

```

'' Example of writing colored text using a sub:

Sub PrintColoredText( ByVal colour As Integer, ByRef
    Color colour
    Print text
End Sub

PrintColoredText( 1, "blue" )           '' a few c
PrintColoredText( 2, "green" )
PrintColoredText( 4, "red" )
Print

Dim i As Integer
For i = 0 To 15                          '' all 1
    PrintColoredText( i, ("color " & i) )
Next i

```

```

' The following demonstrates optional parameters.

Sub TestSub(P As String = "Default")
    Print P
End Sub

TestSub "Testing:"
TestSub

```

Dialect Differences

- The *-lang qb* and *-lang fblite* dialects keep the QB convention default.
- In the *-lang fb* dialect, numeric parameters are passed **ByVal** but passed **ByRef** by default.

Differences from QB

- Public and Private access specifiers are new to FreeBASIC.
- Constructor subroutines are new to FreeBASIC.

See also

- **Declare**
- **Function**
- **Exit**
- **Public**
- **Private**
- **Static**

Sub (Member)



Declares or defines a member procedure.

Syntax

```
{ Type | Class | Union } typename  
Declare [ Static | Const ] Sub fieldname [calling convention specifier] [ Alias external_name ] ( [ parameters ] ) [ Static ]  
End { Type | Class | Union }
```

```
Sub typename.fieldname ( [ parameters ] ) [ Export ]  
statements  
End Sub
```

Parameters

typename

name of the **Type**, **Class**, or **Union**

fieldname

name of the procedure

external_name

name of field as seen when externally linked

parameters

the parameters to be passed to the procedure

calling convention specifier

can be one of: **cdecl**, **stdcall** or **pascal**

Description

sub members are accessed with **Operator** . (**Member Access**) or **Operator** -> (**Pointer To Member Access**) to call a member procedure and may optionally accept parameters either **ByVal** or **ByRef**. *typename* be overloaded without explicit use of the **Overload** keyword.

typename is the name of the type for which the **sub** method is declared and defined. Name resolution for *typename* follows the same rules as procedures when used in a **Namespace**.

A hidden **This** parameter having the same type as *typename* is passed to non-static member procedures. **This** is used to access the fields of

the **Type**, **Class**, or **Union**.

To access duplicated symbols defined outside the Type, use:
.SomeSymbol (Or ..SomeSymbol if inside a **With..End With** block).

A **Static (Member)** may be declared using the **static** specifier. A **const (Member)** may be declared using the **const** specifier.

Example

```
Type Statistics
  count As Single
  sum As Single
  Declare Sub AddValue( ByVal x As Single )
  Declare Sub ShowResults( )
End Type

Sub Statistics.AddValue( ByVal x As Single )
  count += 1
  sum += x
End Sub

Sub Statistics.ShowResults( )
  Print "Number of Values = "; count
  Print "Average           = ";
  If( count > 0 ) Then
    Print sum / count
  Else
    Print "N/A"
  End If
End Sub

Dim stats As Statistics

stats.AddValue 17.5
stats.AddValue 20.1
stats.AddValue 22.3
stats.AddValue 16.9
```

```
stats.ShowResults
```

Output:

```
Number of Values = 4  
Average          = 19.2
```

Dialect Differences

- Only available in the *-lang fb* dialect.

See also

- [Class](#)
- [Function \(Member\)](#)
- [Sub](#)
- [Type](#)

Swap



Exchanges the values of two variables

Syntax

```
Declare Sub Swap ( ByRef a As Any, ByRef b As Any )
```

Parameters

a

A variable to swap.

b

A variable to swap.

Description

Swaps the value of two variables.

Example

```
' using swap to order 2 numbers:  
Dim a As Integer, b As Integer  
  
Input "input a number: "; a  
Input "input another number: "; b  
If a > b Then Swap a, b  
Print "the numbers, in ascending order are:"  
Print a, b
```

Differences from QB

- None

See also

- `Operator = (Assignment)`

Closes all open files and ends the program

Syntax

```
Declare Sub System ( ByVal retval As Long = 0 )
```

Usage

```
System( [ retval ] )
```

Parameters

retval

Error code returned to system.

Description

Closes all open files, exits the program, and returns to the operating system. An optional return value, an integer, can be specified to return an error code to the system. If no return value is given, a value of 0 is automatically returned. This is the same as **End** and is here for compatibility between older BASIC dialects. It is recommended to use **End** instead.

Usage of this statement does not cleanly close scope. Local variables will not have their destructors called automatically, because FreeBASIC does not do stack unwinding. Only the destructors of global variables will be called in this case.

For this reason, it is discouraged to use `system` simply to mark the end of a program; the program will come to an end automatically, and in a cleaner fashion, when the last line of module-level code has executed

Example

```
Print "this text is shown"  
System
```

```
Print "this text will never be shown"
```

Differences from QB

- None

See also

- End

Tab



Sets the column when writing to screen or file

Syntax

```
Tab( col_num )
```

Usage

```
Print Tab( column ) [(, | ;)] ...
```

Parameters

column

1-based column number to move to

Description

Tab will move the cursor to given *column* number when **Printing** to screen or to a file. Character cells skipped over between the old and new cursor positions are left unchanged.

If the current column is greater than *column*, then **Tab** will move the cursor to the requested column number on the next line. If the current column is less than *column*, then the cursor will not move anywhere.

Example

```
' ' Using Print with Tab to justify text in a table

Dim As String A1, B1, A2, B2

A1 = "Jane"
B1 = "Doe"
A2 = "Bob"
B2 = "Smith"

Print "FIRST NAME"; Tab(35); "LAST NAME"
Print "-----"; Tab(35); "-----"
```

```
Print A1; Tab(35); B1
Print A2; Tab(35); B2
```

The output would look like:

FIRST NAME -----	LAST NAME -----
Jane	Doe
Bob	Smith

Differences from QB

- In QBASIC, spaces were printed in the gap, while in FreeBASI characters are just skipped over and left untouched.

See also

- `Spc`
- `Locate`
- `Pos`
- `(Print | ?)`

Tan



Returns the tangent of an angle

Syntax

```
Declare Function Tan ( ByVal angle As Double ) As Double
```

Usage

```
result = Tan( angle )
```

Parameters

angle
the angle (in radians)

Return Value

Returns the tangent of the argument *angle* as a **Double** within the range of negative infinity to positive infinity.

Description

The argument *angle* is measured in **radians** (not **degrees**).

The value returned by this function is undefined for values of *angle* with an absolute value of 2^{63} or greater.

Example

```
Const PI As Double = 3.1415926535897932
Dim a As Double
Dim r As Double
Input "Please enter an angle in degrees: ", a
r = a * PI / 180      'Convert the degrees to Radians
Print ""
Print "The tangent of a" ; a; " degree angle is";
Sleep
```

The output would look like:

```
Please enter an angle in degrees: 75
The tangent of a 75 degree angle Is 3.732050807568878
```

Differences from QB

- None

See also

- [Atn](#)
- [Atan2](#)
- [Sin](#)
- [Cos](#)
- [A Brief Introduction To Trigonometry](#)

Control flow statement for conditional branching.

Syntax

```
If expression Then statement(s) [Else statement(s)]  
or  
If expression Then : statement(s) [Else statement(s)] : End If  
or  
If expression Then  
statement(s)  
[ ElseIf expression Then ]  
statement(s)  
[ Else ]  
statement(s)  
End If
```

Differences from QB

- None

See also

- [If...Then](#)

Hidden instance parameter passed to non-static member functions in a **Type** Or **Class**

Syntax

```
This.fieldname  
or  
With This  
.fieldname  
End With
```

Description

This is a reference to an instance of a **Type** or **Class** that is passed as hidden argument to all non-static member functions of that type or class. Non-static member functions are procedures declared inside the body of a **Type** or **Class** and include **Sub**, **Function**, **Constructor**, **Destructor**, assignment or cast **Operator**, and **Property** procedures.

The **this** additional parameter has the same data type as the **Type** or **Class** in which the procedure is declared.

The **this** parameter can be used just like any other variable, i.e., pass to procedures taking an object of the same type, call other member procedures and access member data using **Operator** **.** (**Member Access**), etc.

Most of the time, using **this** explicitly for member access is unnecessary; member procedures can refer to other members of the instance which they are passed directly by name, without having to qualify it with **This** and **Operator** **.** (**Member Access**). The only times when you need to qualify member names with **this** is when the member name is hidden, for example, by a local variable or parameter. In these situations, qualifying the member name is the only way to refer to these hidden member names.

Example

```
Type sometype
  Declare Sub MyCall()
  value As Integer
End Type

Dim example As sometype

' Set element test to 0
example.value = 0
Print example.value

example.MyCall()

' Output should now be 10
Print example.value

End 0

Sub sometype.MyCall()
  This.value = 10
End Sub
```

Differences from QB

- New to FreeBASIC

See also

- Base
- Class
- Type

Threadcall



Starts a user-defined procedure with parameters in a separate execution

Threadcall uses **LibFFI** internally: people who write programs using this careful to follow LibFFI's license, which can be found at <http://github.com/atgreen/libffi/blob/master/LICENSE>.

Syntax

```
Function Threadcall subname([paramlist]) As Any Ptr
```

Usage

```
threadid = Threadcall subname([paramlist])
```

Parameters

subname

The name of a subroutine

paramlist

A list of parameters to pass to the subroutine, as with a normal sub ca

Return Value

Threadcall returns an **Any Ptr** handle to the thread created, or the nu

Description

Like **ThreadCreate**, **Threadcall** creates a thread which runs at the sam calling it. By placing "**Threadcall**" before almost any normal call to sul inside of a new thread and returns a pointer to that thread.

Using **Threadcall** is simpler method of creating threads, and allows d thread without global variables or pointers which are not type safe. Hc more efficient and should be used for programs creating a large numb

While most subroutines are supported, the following types of subrouti

- Subroutines using **Variable Arguments**
- Subroutines with unions which are passed **ByVal**

- Subroutines with user types containing unions, arrays, s are passed **ByVal**

When using **Threadcall**, parenthesis around the parameter list are required if the subroutine has no parameters.

WARNING: Presently when **Threadcall** involves to pass parameters I cannot guarantee that the corresponding data are still maintained after the `er` statement and this until the thread is launched. That can cause bad behavior.

Example

```

'' Threading using "ThreadCall"

Sub thread( id As String, tlock As Any Ptr, count As Integer)
  For i As Integer = 1 To count
    MutexLock tlock
    Print "thread " & id;
    Locate , 20
    Print i & "/" & count
    MutexUnlock tlock
  Next
End Sub

Dim tlock As Any Ptr = MutexCreate()
Dim a As Any Ptr = ThreadCall thread("A", tlock, 6)
Dim b As Any Ptr = ThreadCall thread("B", tlock, 4)
ThreadWait a
ThreadWait b
MutexDestroy tlock
Print "All done (and without Dim Shared!)"

```

Dialect Differences

- Threading is not allowed in the **-lang qb** dialect.

Platform Differences

- **Threadcall** is not available with the DOS version / target of FreeBASIC. Multithreading is not supported by DOS kernel nor the used execution model.
- In Linux the threads are always started in the order they are created. This is not assumed in Win32. It's an OS, not a FreeBASIC issue.
- In Linux, the **stdcall** and **pascal** calling conventions are not supported.
- In Windows, the **pascal** calling convention is not supported.

Differences from QB

- New to FreeBASIC

See also

- **ThreadCreate**
- **ThreadWait**
- **MutexCreate**
- **MutexLock**
- **MutexUnlock**
- **MutexDestroy**

ThreadCreate



Starts a user-defined procedure in a separate execution thread

Syntax

```
Declare Function ThreadCreate _  
    ( _  
    ByVal procptr As Sub ( ByVal userdata As Any Ptr ), _  
    ByVal param As Any Ptr = 0, _  
    ByVal stack_size As Integer = 0 _  
    ) As Any Ptr
```

Usage

```
result = ThreadCreate ( procptr [, [ param ] [, stack_size ] ] )
```

Parameters

procptr

A pointer to the **Sub** intended to work as a thread. The sub must have same calling convention) to be compatible to *procptr*:

```
Declare Sub myThread ( ByVal userdata As Any Ptr )
```

userdata

The Any Ptr parameter of the **Sub** intended to work as a thread. FreeB must not be omitted!

param

Any Ptr argument that will be passed to the thread **Sub** pointed to by *p* example, this can be a pointer to a structure or an array containing va with. If *param* is not given, 0 (zero) will be passed to the thread sub's *stack_size*

Optional number of bytes to reserve for this thread's stack.

Return Value

ThreadCreate returns an Any Ptr handle to the thread created, or a n

Description

The sub pointed to by *procptr* is started as a thread. It will be passed t specified, in its *userdata* parameter.

The sub that was started as a thread will execute in parallel with the n by assigning it to a different processor if it exists, or by alternating bet There is no guarantee about the order in which different threads exec the order in which multiple create threads actually start executing.

Before closing, programs should wait for the termination of all launche it's not necessary to safely wait for a thread to finish execution, **Thread** exits while some threads are still active, those threads will be aborted programs should call either **ThreadWait** or **Threaddetach** to ensure that thread handles are released. Otherwise, there may be memory or sys

Due to the nature of threads, no assumptions about execution order c between multiple threads, including a thread and the main part of the exclusion locks can be "owned" by a single thread while doing critical turn. See **MutexCreate**, **MutexLock**, **MutexUnlock**, **MutexDestroy**.

stack_size can be used to change the thread's stack size from the sy: program requires a big stack, for example due to lots of procedure rec on the stack. On some systems (Linux), the stack automatically grows on others (Win32), this is the fixed maximum allowed. Behavior is unc reserved size on systems where stacks are not able to grow.

Example

```
' ' Threading synchronization using Mutexes
' ' If you comment out the lines containing "MutexL
' ' the threads will not be in sync and some of the
' ' out of place.

Const MAX_THREADS = 10

Dim Shared As Any Ptr ttylock

' ' Teletype unfurls some text across the screen at
Sub teletype( ByRef text As String, ByVal x As Int
' '
' ' This MutexLock makes simultaneously running
' ' other, so only one at a time can continue a
```

```

'' Otherwise, their Locates would interfere, s
'' cursor.
''
'' It's impossible to predict the order in whi
'' here and which one will be the first to acc
'' causing the rest to wait.
''
MutexLock ttylock

For i As Integer = 0 To (Len(text) - 1)
    Locate x, y + i
    Print Chr(text[i])
    Sleep 25
Next

'' MutexUnlock releases the lock and lets othe
MutexUnlock ttylock
End Sub

Sub thread( ByVal userdata As Any Ptr )
    Dim As Integer id = CInt(userdata)
    teletype "Thread (" & id & ").....", 1 + i
End Sub

'' Create a mutex to synchronize the threads
ttylock = MutexCreate()

'' Create child threads
Dim As Any Ptr handles(0 To MAX_THREADS-1)
For i As Integer = 0 To MAX_THREADS-1
    handles(i) = ThreadCreate(@thread, CPtr(Ar
    If handles(i) = 0 Then
        Print "Error creating thread:"; i
        Exit For
    End If
Next

'' This is the main thread. Now wait until all
For i As Integer = 0 To MAX_THREADS-1

```

```

        If handles(i) <> 0 Then
            ThreadWait(handles(i))
        End If
    Next

    '' Clean up when finished
    MutexDestroy(ttylock)

```

```

Sub print_dots(ByRef char As String)
    For i As Integer = 0 To 29
        Print char;
        Sleep CInt(Rnd() * 100), 1
    Next
End Sub

Sub mythread(param As Any Ptr)
    '' Work (other thread)
    print_dots("*")
End Sub

Randomize(Timer())

Print " main thread: ."
Print "other thread: *"

'' Launch another thread
Dim As Any Ptr thread = ThreadCreate(@mythread)

'' Work (main thread)
print_dots(".")

'' Wait until other thread has finished, if ne
ThreadWait(thread)
Print
Sleep

```

```

'' Threaded consumer/producer example using mutexes

Dim Shared As Any Ptr produced, consumed

Sub consumer( ByVal param As Any Ptr )
    For i As Integer = 0 To 9
        MutexLock produced
        Print ", consumer gets:", i
        Sleep 500
        MutexUnlock consumed
    Next
End Sub

Sub producer( ByVal param As Any Ptr )
    For i As Integer = 0 To 9
        Print "Producer puts:", i;
        Sleep 500
        MutexUnlock produced
        MutexLock consumed
    Next i
End Sub

Dim As Any Ptr consumer_id, producer_id

produced = MutexCreate
consumed = MutexCreate
If( ( produced = 0 ) Or ( consumed = 0 ) ) Then
    Print "Error creating mutexes! Exiting..."
    End 1
End If

MutexLock produced
MutexLock consumed
consumer_id = ThreadCreate(@consumer)
producer_id = ThreadCreate(@producer)
If( ( producer_id = 0 ) Or ( consumer_id = 0 ) )

```

```
Print "Error creating threads! Exiting..."
End 1
End If

ThreadWait consumer_id
ThreadWait producer_id

MutexDestroy consumed
MutexDestroy produced

Sleep
```

Dialect Differences

- Threading is not allowed in the *-lang qb* dialect.

Platform Differences

- **Threadcreate** is not available with the DOS version / target of supported by DOS kernel nor the used extender.
- In Linux the threads are always started in the order they are created, not a FreeBASIC issue.

Differences from QB

- New to FreeBASIC

See also

- **ThreadWait**
- **Threaddetach**
- **MutexCreate**
- **MutexLock**
- **MutexUnlock**
- **MutexDestroy**

Threaddetach



Releases a thread handle without waiting for the thread to finish

Syntax

```
Declare Sub ThreadDetach ( ByVal id As Any Ptr )
```

Usage

```
#include "fbthread.bi"  
ThreadDetach( id )
```

Parameters

id

Any Ptr handle of a thread created by **ThreadCreate** or **Threadcall**

Description

ThreadDetach releases resources associated with a thread handle returned by **ThreadCreate** or **Threadcall**. The thread handle will be destroyed by **ThreadDetach** and cannot be used anymore.

Unlike **Threadwait**, **ThreadDetach** does not wait for the thread to finish and thread execution continues independently. Any allocated resources will be freed once the thread exits.

Example

```
#include "fbthread.bi"  
  
Sub mythread( ByVal param As Any Ptr )  
    Print "hi!"  
End Sub  
  
Var thread = ThreadCreate( @mythread )  
threaddetach( thread )  
  
threaddetach( ThreadCreate( @mythread ) )
```

Sleep

Dialect Differences

- Threading is not allowed in the *-lang qb* dialect.

Platform Differences

- **ThreadDetach** is not available with the DOS version of FreeBASIC, because multithreading is not supported by DOS kernel nor the used extender.

Differences from QB

- New to FreeBASIC

See also

- [ThreadWait](#)
- [ThreadCreate](#)

ThreadWait



Waits for a thread to finish execution and releases the thread handle

Syntax

```
Declare Sub ThreadWait ( ByVal id As Any Ptr )
```

Usage

```
ThreadWait( id )
```

Parameters

id

Any Ptr handle of a thread created by `ThreadCreate` or `Threadcall`

Description

ThreadWait waits for a thread created by `ThreadCreate` or `Threadcall` to finish execution, and then releases the resources associated with the thread handle. **ThreadWait** does not return until the thread designated by *id* ends.

In order to release a thread handle without waiting for the thread to finish, use `Threaddetach`.

ThreadWait does not force the thread to end; if a thread requires a signal to force its end, a mechanism such as shared variables and mutexes must be used.

Example

See the `ThreadCreate` examples.

Dialect Differences

- Threading is not allowed in the `-lang qb` dialect.

Platform Differences

- **ThreadWait** is not available with the DOS version of FreeBASIC, because multithreading is not supported by DOS kernel nor the used extender.

Differences from QB

- New to FreeBASIC

See also

- [ThreadCreate](#)
- [Threaddetach](#)

Time



Returns the current system time as a string

Syntax

```
Declare Function Time ( ) As String
```

Usage

```
result = Time
```

Return Value

Returns the current system.

Description

Returns the current system time in the format hh:mm:ss.

Example

```
Print "the current time is: "; Time
```

Differences from QB

- The QB TIME statement (to set the system time) is now called `SetTime`.

See also

- `Date`
- `Timer`

TimeSerial



Gets a **Date Serial** for the specified hours, minutes, and seconds

Syntax

```
Declare Function TimeSerial ( ByVal hour As Long, ByVal minute As Long, ByVal second As Long ) As Double
```

Usage

```
#include "vbcompat.bi"  
result = TimeSerial( hours, minutes, seconds )
```

Parameters

hour
number of hours, in the range 0-23

minute
number of minutes

second
number of seconds

Return Value

Returns a **date serial** containing the time formed by the values in the parameters. The date serial returned has no integer part.

Description

hours must be specified in the range 0-23

The compiler will not recognize this function unless `vbcompat.bi` or `dateutil.bi` is included.

Example

```
#include "vbcompat.bi"  
  
Dim ds As Double = DateSerial(2005, 11, 28) + TimeSerial(12, 30, 30)
```

```
Print Format(ds, "yyyy/mm/dd hh:mm:ss")
```

Differences from QB

- Did not exist in QB. This function appeared in PDS and VBDO:

See also

- **Date Serials**
- **DateSerial**
- **TimeValue**
- **DateValue**

TimeValue



Gets a **Date Serial** from a time string

Syntax

```
Declare Function TimeValue ( ByRef timestring As String ) As Double
```

Usage

```
#include "vbcompat.bi"  
result = TimeValue( timestring )
```

Parameters

timestring
the string to convert

Return Value

Returns a **Date Serial** from a time string.

Description

The time string must be in the format "23:59:59" or "11:59:59PM"

The compiler will not recognize this function unless `vbcompat.bi` or `datetime.bi` is included.

Example

```
#include "vbcompat.bi"  
  
Dim ds As Double = TimeValue("07:12:28AM")  
  
Print Format(ds, "hh:mm:ss")
```

Differences from QB

- Did not exist in QB. This function appeared in PDS and VBDO:

See also

- **Date Serials**
- **DateSerial**
- **TimeValue**
- **DateValue**

Returns the amount of time that has passed since a static reference poi

Syntax

Declare Function Timer () As Double

Usage

result = Timer

Return Value

Returns a **Double** precision result with the time, in seconds, since a st reference point.

Description

The `Timer` function is useful for finding out how long a section of code to run, or for control the timing of your code. To find out how much tim passed between two points in your program, you can record the value `Timer` at the start and end points, and then subtract the start value from end value.

On some platforms, the value of `Timer` resets to zero at midnight (see so if the start and end time are on either side of the reset point, the dif will be negative. This could cause unexpected behavior in some progr those cases, adding 86400 (the number of seconds in 24 hours) to the difference should return the correct result. If the time taken is longer th day, then it will be also be necessary to check the number of days tha elapsed.

The value returned by `Timer` is NOT affected by the automatic changir the system clock, in Spring and Autumn, for DST (Daylight Savings Ti

Example

```

'' Example of using TIMER function
'' Note: see text about correct waiting strategies
Dim Start As Double
Print "Wait 2.5 seconds."
Start = Timer
Do
    Sleep 1, 1
Loop Until (Timer - Start) > 2.5
Print "Done."

```

Platform Differences

- On Win32 and Linux, if the program must wait for periods of 0. seconds or more, **sleep** should be used, this allows other programs to run during the waiting period. For shorter delays, a loop using **sleep** can be more precise.
- The reference point chosen varies, depending on the platform. On Windows, the time is measured relative to the point the computer was booted up. On DOS, the time is measured relative to Jan 1 1970.

*Note for DOS users: today, the number of seconds since 1970 is in excess of 10^9 , and is therefore unsuitable for storing in **single**-precision variables. It shouldn't be multiplied (to get 1/10 seconds or so) and stored in 32-bit integer variables then*

- The precision of TIMER varies, depending on the computer used. If the processor has a precision timer (as the Performance Counter Pentium processors from Intel have) and the OS uses it, the precision is linked to the processor clock and microseconds can be expected. With older processors (386, 486), and always in DOS, the resolution is 1/18 second.
- Usage of TIMER can cause disk accesses in DOS, see **forum** analysis and solutions

Differences from QB

- In QB, TIMER returned the number of seconds from last midnight, its accuracy was 1/18 secs

See also

- [Time](#)
- [Sleep](#)

Statement modifier to specify a range.

Syntax

```
For iterator initial_value To ending_value
statement(s).
Next [ iterator ]
or
Select Case case_comparison_value
Case lower_bound To upper_bound
statement(s).
End Select
or
Dim variable_identifier( lower_bound To upper_bound ) As type_sp
```

Description

The **to** keyword is used to define a certain numerical range. This keyw

In the first syntax, the **to** keyword defines the initial and ending values

In the second syntax, the **to** keyword defines lower and upper bounds

In the third syntax, the **to** keyword defines the array bounds in a **Dim** s

For more information, see **For...Next**, **Dim** and **Select Case**.

Example

```
' ' this program uses bound variables along with th
' ' temperatures inside the array, and to determine
Randomize Timer

' ' define minimum and maximum number of temperatur
Const minimum_temp_count As Integer = 1
Const maximum_temp_count As Integer = 10

' ' define the range of temperatures zones in which
```

```

Const min_low_danger As Integer = 40
Const max_low_danger As Integer = 69
Const min_medium_danger As Integer = 70
Const max_medium_danger As Integer = 99
Const min_high_danger As Integer = 100
Const max_high_danger As Integer = 130

'' define array to hold temperatures using our mir
Dim As Integer array( minimum_temp_count To maximum_t

'' declare a for loop that iterates from minimum t
Dim As Integer it
For it = minimum_temp_count To maximum_temp_count

    array( it ) = Int( Rnd( 1 ) * 200 ) + 1

'' display a message based on temperature using
Select Case array( it )
    Case min_low_danger To max_low_danger
        Color 11
        Print "Temperature" ; it ; " is in the lo
    Case min_medium_danger To max_medium_danger
        Color 14
        Print "Temperature" ; it ; " is in the me
    Case min_high_danger To max_high_danger
        Color 12
        Print "Temperature" ; it ; " is in the hi
    Case Else
        Color 3
        Print "Temperature" ; it ; " is safe at"
End Select

Next it

Sleep

```

Differences from QB

- none

See also

- [For...Next](#)
- [Dim](#)
- [Select Case](#)

Trans



Parameter to the `Put` graphics statement which selects transparent back

Syntax

```
Put [ target, ] [ STEP ] ( x,y ), source [ ,( x1,y1 )-( x2,y2 )
```

Parameters

`Trans`
Required.

Description

`Trans` selects transparent background as the method for blitting an image. For 8-bit color images, the mask color is palette index 0. For 16/32-bit images, the mask color is the alpha value of pixels in 32-bit images.

Note: for 32-bit images, the alpha value of pixels may be changed to (example below).

Example

```
' set up a screen: 320 * 200, 16 bits per pixel
ScreenRes 320, 200, 16

' set up an image with the mask color as the back
Dim img As Any Ptr = ImageCreate( 32, 32, RGB(255,
Circle img, (16, 16), 15, RGB(255, 255, 0),
Circle img, (10, 10), 3, RGB( 0, 0, 0),
Circle img, (23, 10), 3, RGB( 0, 0, 0),
Circle img, (16, 18), 10, RGB( 0, 0, 0), 3.14,

' Put the image with PSET (gives the exact center
Draw String (110, 50 - 4), "Image put with PSET"
Put (60 - 16, 50 - 16), img, PSet

' Put the image with TRANS
```

```

Draw String (110, 150 - 4), "Image put with TRANS"
Put (60 - 16, 150 - 16), img, Trans

'' free the image memory
ImageDestroy img

'' wait for a keypress
Sleep

```

```

Function trans32 ( ByVal source_pixel As UInteger,
'' returns the source pixel
'' unless it is &hff00ff (magenta), then return
If (source_pixel And &hfffffff) <> &hff00ff Then
Return source_pixel
Else
Return destination_pixel
End If
End Function

'' set up a screen: 320 * 200, 16 bits per pixel
ScreenRes 320, 200, 32

'' set up an image with the mask color as the back
Dim img As Any Ptr = ImageCreate( 32, 32, RGB(255,
Circle img, (16, 16), 15, RGB(255, 255, 0),
Circle img, (10, 10), 3, RGB( 0, 0, 0),

```

```

Circle img, (23, 10), 3, RGB( 0, 0, 0),
Circle img, (16, 18), 10, RGB( 0, 0, 0), 3.14,

'' Put the image with PSET (gives the exact center)
Draw String (110, 50 - 4), "Image put with PSET"
Put (60 - 16, 50 - 16), img, PSet

'' Put the image with TRANS
Draw String (110, 100 - 4), "Image put with TRANS"
Put (60 - 16, 100 - 16), img, Trans

'' Put the image with TRANS
Draw String (110, 150 - 4), "Image put with trans32"
Put (60 - 16, 150 - 16), img, Custom, @trans32

'' free the image memory
ImageDestroy img

'' wait for a keypress
Sleep

```

Differences from QB

- New to FreeBASIC

See also

- [Put \(Graphics\)](#)
- [Custom](#)

Removes surrounding substrings or characters on the left and right side string

Syntax

```
Declare Function Trim ( ByRef str As Const String, [ Any ] ByRef  
trimset As Const String = " " ) As String  
Declare Function Trim ( ByRef str As Const WString, [ Any ] ByRe  
trimset As Const WString = WStr(" ") ) As WString
```

Usage

```
result = Trim[$]( str [, [ Any ] trimset ] )
```

Parameters

str

The source string.

trimset

The substring to trim.

Return Value

Returns the trimmed string.

Description

This procedure trims surrounding characters from the left (beginning) right (end) of a source string. Substrings matching *trimset* will be trimmed, otherwise spaces (**ASCII** code 32) are trimmed.

If the **Any** keyword is used, any character matching a character in *trimset* will be trimmed.

All comparisons are case-sensitive.

Example

```

Dim s1 As String = " ... Stuck in the middle ... "
Print "" + Trim(s1) + ""
Print "" + Trim(s1, Any ".") + ""

Dim s2 As String = "BaaBaaaaB With You aaBBaaBaa"
Print "" + Trim(s2, "Baa") + ""
Print "" + Trim(s2, Any "Ba") + ""

```

will produce the output:

```

'... Stuck in the middle ...'
'Stuck in the middle'
'aaB With You aaB'
' With You '

```

Platform Differences

- DOS version/target of FreeBASIC does not support the wide-c version of `Trim`.

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the `__Trim`.
- The string type suffix "\$" is optional in the *-lang fblite* and *-lan* dialects.

Differences from QB

- New to FreeBASIC

See also

- [LTrim](#)
- [RTrim](#)

True



Intrinsic constant set by the compiler

Syntax

```
Const True As Boolean
```

Description

Gives the True `Boolean` value where used.

Example

```
Dim b As Boolean = True
If b Then
    Print "b is True"
Else
    Print "b is False"
End If
```

```
b is True
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__True`.

Differences from QB

- New to FreeBASIC

See also

- **False**
- **Boolean**

Type (Alias)



Declares an alternative name for a type

Syntax

```
Type typename As symbol
```

Parameters

typename

new alternative name.

symbol

symbol or data type declaration to associate with *typename*.

Description

symbol may refer to any declared data type including a built-in data type, **Sub** or **Function** pointer, **Type** declaration, **Union** declaration, or **Enum** declaration.

A type alias can be used to allow forward declarations of parameters in procedure declarations, but only used with pointers or parameters passed by reference (excluding arrays).

A type alias can also be used to allow forward declarations of data fields in **User Defined Types**, but only used with pointers.

Example

```
Type ParentFwd As Parent
Type Child
  Name As ZString * 32
  ParentRef As ParentFwd Ptr
  ' ' . . .
End Type

Type Parent
  Name As ZString * 32
```

```

        ChildList(0 To 9) As Child
        '' ...
End Type

Dim p As Parent
p.Name = "Foo"
With p.ChildList(0)
    .Name = "Jr."
    .ParentRef = @p
    '' ...
End With

With p.ChildList(0)
    Print .Name; " is child of "; .parentRef-
>Name
End With

```

Differences from QB

- New to FreeBASIC

See also

- Type...End Type
- Type (Temporary)

Temporary Types



Creates a temporary copy of a user defined type

Syntax

```
result = Type( initializers, ... )  
or  
result = Type<typename>( initializers, ... )
```

Parameters

initializers
Initial values for the type
typename
The name of the **Type** or **Union**

Return Value

A temporary copy of the type.

Description

Used to create a temporary type. If *typename* is not explicitly given, it will be inferred from the context.
Usage of the temporary copy may include assigning it to a variable, passing it to a function, or returning it as a value from a procedure.

For a type without constructor, the temporary type syntax is allowed for primitives only and without any default initializers, but the compiler does not create a temporary copy if at the same time the type is without destructor.

The **Constructor** for the type, if there is one, will be called when the temporary type is created. The **Destructor** for the type, if there is one, will be called immediately after the temporary type expression. The temporary type expression may be simply replaced by *typename*(*initializers*, ...) if the type has a constructor and a destructor.

It can create not only a temporary copy of an user defined type, but also a temporary copy of a primitive type as a variable-length string or any numeric data-type (all standard strings).

It can also be used as an even shorter shortcut than **with** (see below)

Example

```
Type Example
    As Integer field1
    As Integer field2
End Type

Dim ex As Example

' Filling the type by setting each field
ex.field1 = 1
ex.field2 = 2

' Filling the type by setting each field using WI
With ex
    .field1 = 1
    .field2 = 2
End With

' Fill the variable's fields with a temporary ty
ex = Type( 1, 2 )
```

```
' Passing a user-defined types to a procedure usi
' where the type can be inferred.

Type S
    As Single x, y
End Type

Sub test ( v As S )
    Print "S", v.x, v.y
End Sub

test( Type( 1, 2 ) )
```

```

'' Passing a user-defined type to a procedure using
'' where the type is ambiguous and the name of the

Type S
  As Single x, y
End Type

Type T
  As Integer x, y
End Type

Union U
  As Integer x, y
End Union

'' Overloaded procedure test()
Sub test Overload ( v As S )
  Print "S", v.x, v.y
End Sub

Sub test ( v As T )
  Print "T", v.x, v.y
End Sub

Sub test ( v As U )
  Print "U", v.x, v.y
End Sub

'' Won't work: ambiguous
'' test( type( 1, 2 ) )

'' Specify name of type instead
test( Type<S>( 1, 2 ) )
test( Type<T>( 1, 2 ) )
test( Type<U>( 1 ) )

```

Differences from QB

- New to FreeBASIC

See also

- [Type...End Type](#)
- [Type \(Alias\)](#)

Declares a user-defined type.

Syntax

```
Type typename  
  fieldname1 As DataType  
  fieldname2 As DataType  
  As DataType fieldname3, fieldname4  
  ...  
End Type
```

```
Type typename [Extends base_typename] [Field = alignment]  
  [Private:|Public:|Protected:]
```

```
Declare Sub|Function|Constructor|Destructor|Property|Operator ..  
Static variablename As DataType
```

```
fieldname As DataType [= initializer]  
fieldname(array dimensions) As DataType [= initializer]  
fieldname(Any [, Any...]) As DataType  
fieldname : bits As DataType [= initializer]
```

```
As DataType fieldname [= initializer], ...  
As DataType fieldname(array dimensions) [= initializer], ...  
As DataType fieldname(Any [, Any...])  
As DataType fieldname : bits [= initializer], ...
```

Union

```
fieldname As DataType  
Type  
  fieldname As DataType  
  ...  
End Type  
  ...  
End Union
```

```
...  
End Type
```

Description

Type is used to declare custom data types containing one or more data variable-length (dynamic) arrays, fixed-size or variable-length strings,

Types support various functionality related to object-oriented program

- Inheritance through the use of the **Extends** keyword
- Member procedures such as **Subs** or **Functions**, includ
- Member procedures with special semantic meaning suc
- **Static** member variables
- Member visibility specifiers: **Public:**, **Private:**, **Protecte**

Types may also contain nested types or unions, allowing data membe contain member procedures or static member variables (same restrict

Memory layout

Types lay out their fields consecutively in memory, following the native care must be taken when using Types for file I/O or interacting with ot padding rules are different. The optional **Field = number** specifier can

Variable-length data

In FreeBASIC, Type data structures must ultimately be fixed-size, suc that Type. Nevertheless, Types may contain variable-length (dynamic) not be embedded in the Type directly. Instead, the Type will only conta the scenes to manage the variable-length string/array data. For sizing (dynamic) array data member must be always declared by using **Any(:** dimensions based on the number of Anys specified.

Because of that, saving such a Type into a file will write out the descri into Types directly, fixed-length strings/arrays must be used.

Similarly, when maintaining dynamic data manually through the use o Type to a file, because the address stored in the pointer field will be w meaningful to a specific process only though, and cannot be shared th

Special note on fixed-length strings

Currently, fixed-length string fields of **String * N** type have an extra n incompatible with QB strings inside Types, because they actually use declare the field As **String * (N-1)**, though this will not work in future a **Byte** or **UByte** array with the proper size.

Example

This is an example of a QB-style type, not including procedure definiti

```
Type clr
  red As UByte
  green As UByte
  blue As UByte
End Type
```

```
Dim c As clr
c.red = 255
c.green = 128
c.blue = 64
```

And this is an example of a type working as an object:

```
' ' Example showing the problems with fixed length
' ' Suppose we have read a GIF header from a file
' '
' ' signature width
Dim As ZString*(10+1) z => "GIF89a" + MKShort(10)

Print "Using fixed-length string"

Type hdr1 Field = 1
  As String*(6-1) sig /' We have to dimension the
  ' less to avoid misalignment
  As UShort wid, hei
End Type

Dim As hdr1 Ptr h1 = CPtr(hdr1 Ptr, @z)
Print h1->sig, h1->wid, h1->hei ' ' Prints GIF89 (m

' ' We can do comparisons only with the 5 visible c
If Left(h1->sig, 5) = "GIF89" Then Print "ok" Else

' ' Using a ubyte array, we need an auxiliary funct
```

```

Function ub2str( ub() As UByte ) As String
    Dim As String res = Space(UBound(ub) - LBound(
    For i As Integer = LBound(ub) To UBound(ub)
        res[i - LBound(ub)] = ub(i)
    Next
    Function = res
End Function

Print
Print "Using an array of ubytes"

Type hdr2 Field = 1
    sig(0 To 6-1) As UByte '' Dimension 6
    As UShort wid, hei
End Type

Dim As hdr2 Ptr h2 = CPtr(hdr2 Ptr, @z)
'' Viewing and comparing is correct but a conversi

Print ub2str(h2->sig()), h2->wid, h2->hei '' Print
If ub2str(h2->sig()) = "GIF89a" Then Print "ok" El

```

Platform Differences

- The default `Field` alignment parameter is 4 bytes for DOS and
- The default `Field` alignment parameter is 8 bytes for Windows and Double members).

Dialect Differences

- Object-related features such as functions declared inside `Type`
- In the *-lang fb* and *-lang fblite* dialects, the default `Field` align
- With the *-lang qb* dialect the fields are aligned to byte boundar
- To force byte alignment use `FIELD=1`.

Differences from QB

- At present, fixed-length strings have an extra, redundant character in QB. For this reason, UDTs that use them are not compatible

See also

- [Type \(Alias\)](#)
- [Type \(Temporary\)](#)
- [Union](#)
- [Enum](#)
- [TypeOf](#)
- [OffsetOf](#)
- [Field](#)
- [Extends](#)
- [With](#)

TypeOf



Returns the type of a variable.

Syntax

TypeOf (*variable* | *datatype*)

Parameters

variable

A variable of any type.

datatype

A **DataType**.

Description

TypeOf is a compiler intrinsic that replaces itself with the type of the variable in a variable declaration (Example 1) or it can be used in the preprocessor (Example 2)

TypeOf also supports passing any intrinsic data type, or user-defined types. Also supported are expressions, the type is inferred from the expression.

If there is both a user defined type and a variable visible with the same name, the user defined type takes precedence over the variable. To ensure that the user defined type is used, wrap the argument to **TypeOf** with parentheses to force the compiler to use the user defined type. For example `TypeOf((variable))`.

Example

Example 1:

```
Dim As Integer foo
Dim As TypeOf(67.2) bar ' '67.2' is a literal double
Dim As TypeOf( foo + bar ) teh_double ' double + integer
Print SizeOf(teh_double)
```

Example 2:

```
Dim As String foo
#print TypeOf(foo)
#if TypeOf(foo) = TypeOf(Integer)
    #print "Never happened!"
#endif

#if TypeOf(foo) = TypeOf(String)
    #print "It's a String!"
#endif
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- [SizeOf](#)
- [Var](#)
- [Type \(Alias\)](#)
- [Type...End Type](#)

UBound



Returns the upper bound of an array's dimension

Syntax

```
Declare Function UBound ( array() As Any, ByVal dimension As Int
```

Usage

```
result = UBound( array [, dimension ] )
```

Parameters

array

an array of any type

dimension

the dimension to get upper bound of

Return Value

Returns the upper bound of an array's dimension.

Description

UBound returns the largest value that can be used as an index into a p

Array dimensions are numbered from one (1) to n , where n is the total will return the upper bound of the first dimension.

If *dimension* is zero (0), **UBound** returns n , the number of dimensions in valid range $1..n$, the result is -1. This can be used to detect the number combination with the result of **Lbound**() for such cases, whether a give dimensions). See the **LBound** page for more information.

Example

```
Dim array(-10 To 10, 5 To 15, 1 To 2) As Integer
```

```
Print UBound(array) 'returns 10
Print UBound(array, 2) 'returns 15
Print UBound(array, 3) 'returns 2
```

```
' ' determining the size of an array
Dim As Short array(0 To 9)
Dim As Integer arraylen, arraysize

arraylen = UBound(array) - LBound(array) + 1
arraysize = arraylen * SizeOf( Short )

Print "Number of elements in array:", arraylen
Print "Number of bytes used in array:", arraysize
```

```
' ' determining the size of a multi-dimensional array
Dim As Long array4D(1 To 2, 1 To 3, 1 To 4, 1 To 5)
Dim As Integer arraylen, arraysize

arraylen = (UBound(array4D, 4) - LBound(array4D, 4) + 1)
          * (UBound(array4D, 3) - LBound(array4D, 3) + 1)
          * (UBound(array4D, 2) - LBound(array4D, 2) + 1)
          * (UBound(array4D, 1) - LBound(array4D, 1) + 1)

arraysize = arraylen * SizeOf( Long )

Print "Number of elements in array:", arraylen
Print "Number of bytes used in array:", arraysize
```

```
' ' determining whether an array is empty
Dim array() As Integer
```

```

Print "lbound: "; LBound( array ), "ubound: "; UBound( array )

If LBound( array ) > UBound( array ) Then
    Print "array is empty"
Else
    Print "array is not empty"
End If

```

```

Sub printArrayDimensions( array() As Integer )
    Print "dimensions: " & UBound( array, 0 )

    ' For each dimension...
    For d As Integer = LBound( array, 0 ) To UBound( array, 0 )
        Print "dimension " & d & ": " & LBound( array, d )
    Next
End Sub

Dim array() As Integer
printArrayDimensions( array() )

Print "---"

ReDim array(10 To 11, 20 To 22)
printArrayDimensions( array() )

```

See also

- [LBound](#)
- [Static](#)
- [Dim](#)
- [ReDim](#)
- [SizeOf](#)

UByte



Standard data type: 8 bit unsigned

Syntax

```
Dim variable As UByte
```

Description

8-bit unsigned whole-number data type. Can hold a value in the range of 0 to 255.

Example

```
Dim ubytevar As UByte
ubytevar = 200
Print "ubytevar= ", ubytevar
```

Example

```
Dim x As UByte = 0
Dim y As UByte = &HFF
Print "UByte Range = "; x; " to "; y
```

Output:

```
UByte Range = 0 to 255
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__ubyte`.

Differences from QB

- New to FreeBASIC

See also

- [Byte](#)
- [CUByte](#)

Returns an upper case copy of a string

Syntax

```
Declare Function UCase ( ByRef str As Const String, ByVal mode As Long = 0 ) As String
Declare Function UCase ( ByRef str As Const WString, ByVal mode As Long = 0 ) As WString
```

Usage

```
result = UCase[$]( str [ , mode ] )
```

Parameters

str

String to convert to uppercase.

mode

The conversion mode: 0 = current locale, 1 = ASCII only

Return Value

Uppercase copy of *str*.

Description

Returns a copy of *str* with all of the letters converted to upper case.

If *str* is empty, the null string ("") is returned.

Example

```
Print UCase("AbCdEfG")
```

will produce the output:

ABCDEFGG

Platform Differences

- The wide-character string version of `ucase` is not supported for DOS target.

Dialect Differences

- The string type suffix "\$" is obligatory in the *-lang qb* dialect.
- The string type suffix "\$" is optional in the *-lang fblite* and *-lang fb* dialects.

Differences from QB

- QB does not support Unicode.

See also

- `LCASE`

UInteger



Standard data type: 32-bit or 64-bit unsigned, same size as `sizeof(Any Ptr)`

Syntax

```
Dim variable As UInteger
Dim variable As UInteger<bits>
```

Parameters

bits

A numeric constant expression indicating the size in bits of unsigned integer desired. The values allowed are 8, 16, 32 or 64.

Description

32-bit or 64-bit unsigned whole-number data type, depending on the platform.

If an explicit bit size is given, a data type is provided that can hold values from 0 up to `(1ULL shl (bits)) - 1`.

Example

```
#if __FB_64BIT__
  Dim x As UInteger = 0
  Dim y As UInteger = &HFFFFFFFFFFFFFFFF
  Print "UInteger Range = "; x; " to "; y
#else
  Dim x As UInteger = 0
  Dim y As UInteger = &HFFFFFFFF
  Print "UInteger Range = "; x; " to "; y
#endif
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__UInteger`.

Differences from QB

- New to FreeBASIC

See also

- [Integer](#)
- [Unsigned](#)
- [CUInt](#)

Ulong



Standard data type: 32-bit unsigned integer

Syntax

```
Dim variable As Ulong
```

Description

32-bit unsigned whole-number data type. Can hold values from 0 to 4294967295. Corresponds to an unsigned DWORD.

Example

```
Dim x As ULong = 0
Dim y As ULong = &HFFFFFFF
Print "ULong Range = "; x; " to "; y
```

Output:

```
ULong Range = 0 to 4294967295
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__ulong`.

Differences from QB

- New to FreeBASIC

See also

- [Long](#)
- [UInteger](#)

- `ULongInt`

ULongInt



Standard data type: 64 bit unsigned

Syntax

```
Dim variable As ULongInt
```

Description

A 64-bit unsigned whole-number data type. Can hold values from 0 to 18 446 744 073 709 551 615. Corresponds to an unsigned QWORD.

Example

```
Dim x As ULongInt = 0
Dim y As ULongInt = &HFFFFFFFFFFFFFFFFull
Print "ULongInt Range = "; x; " to "; y
```

Output:

```
ULongInt Range = 0 to 18446744073709551615
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__Ulongint`.

Differences from QB

- New to FreeBASIC

See also

- [LongInt](#)

- **CULngInt**

Declares a union user defined type.

Syntax

```
Union typename  
fieldname as datatype  
Declare member function declaration ...  
...  
End Union
```

Parameters

typename
Name of the **Union**

fieldname
Name of a data field member

member function declaration
Any of the supported member functions

Description

Unions are similar to a **Type** structure, except that the elements of a union share the same space in memory.

Like Type, Union can use the optional **Field** = *number* specifier and subscripts through the use of the **Extends** keyword.

Unlike Type, Union can not contain variable-length strings, and more (and does not have bases) with constructors or destructors.

The size of the Union is the size of the largest data item. A data item cannot be used since they occupy the same space, only a single element can be used.

Unions support member functions including **Constructor**, **Destructor**, **Property** and **Sub**. All members of a union are public and access control is not applicable.

Nested unnamed type or union cannot have procedure members or subroutines (same restriction for local named type/union).

A **Union** can be passed as a user defined type to overloaded operator

Example

```
' Example 1: bitfields.
Type unitType
  Union
    Dim attributeMask As UInteger
    Type      ' 32-bit uintegers can support up to 32
      isMilitary      : 1 As UInteger
      isMerchant      : 1 As UInteger
    End Type
  End Union
End Type

Dim myunit As unitType
myunit.isMilitary = 1
myunit.isMerchant = 1
Print myunit.isMilitary      ' Result: 1.
Print myunit.isMerchant     ' Result: 1.
Print myunit.attributeMask  ' Result: 3.
Sleep

' Example 2.
' Define our union.
Union AUnion
  a As UByte
  b As Integer
End Union
' Define a composite type.
Type CompType
  s As String * 20
  ui As Byte 'Flag to tell us what to use in uni
  Union
    au As UByte
    bu As Integer
  End Union
End Type

' Flags to let us know what to use in union.
```

```

' You can only use a single element of a union.
Const IsInteger = 1
Const IsUByte = 2

Dim MyUnion As AUnion
Dim MyComposite As CompType

' Can only set one value in union.
MyUnion.a = 128

MyComposite.s = "Type + Union"
MyComposite.ui = IsInteger ' Tells us this is an i
MyComposite.bu = 1500

Print "Union: ";MyUnion.a

Print "Composite: ";
If MyComposite.ui = IsInteger Then
    Print MyComposite.bu
ElseIf MyComposite.ui = IsUByte Then
    Print MyComposite.au
Else
    Print "Unknown type."
End If

Sleep

```

Dialect Differences

- Object-related features as functions defined inside the **union** block are only available in the *-lang fb* dialect.
- Not available in the *-lang qb* dialect unless referenced with the `fb` keyword.

Differences from QB

- New to FreeBASIC

See also

- [Type](#)

Removes a previous access restriction (lock) on a file

Syntax

```
Unlock #filenum, record  
Unlock #filenum, start To end
```

Parameters

filenum

The file number used to **open** the file.

record

The record (**Random** files) to unlock.

start

The first byte position (**Binary** files) in a range to unlock.

end

The last byte position (**Binary** files) in a range to unlock.

Description

Unlock removes the temporary access restriction set by **Lock**.

It is strongly recommended to use the same arguments used in the previous **Lock**.

Note: This command does not always work, neither as documented nor as expected. It appears to be broken at the moment.

Example

For an example see **Lock**.

Differences from QB

- Currently, FB cannot implicitly unlock the entire file
- In **Random** mode, FB cannot unlock a range of records

See also

- **Lock**
- **Open**
- **ScreenUnlock**

Unsigned



Integer data type modifier

Syntax

```
Dim variable As Unsigned {integer-based data type}
```

Description

Forces an integer-based data type to be unsigned (cannot contain negative numbers, but has its maximum value doubled).

Example

```
'e.g. notice what is displayed:
```

```
Dim x As Unsigned Integer  
x = -1  
Print x
```

```
'output is 4294967295
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__Unsigned`.

Differences from QB

- New to FreeBASIC

See also

- `UInteger`

Until



Conditional clause used in **Do . . Loop** statements.

Syntax

```
Do Until condition  
or  
Loop Until condition
```

Description

until is used with the **Do . . Loop** structure.

Example

```
Dim a As Integer  
  
a = 1  
Do  
    Print "hello"  
a = a + 1  
Loop Until a > 10
```

'This will continue to print "hello" on the screen

Differences from QB

- None

See also

- **Do . . Loop**

UShort



Standard data type: 16 bit unsigned

Syntax

```
Dim variable As UShort
```

Description

16-bit unsigned whole-number data type. Can hold values from 0 to 65535.

Example

```
Dim x As UShort = 0
Dim y As UShort = &HFFFF
Print "UShort Range = "; x; " to "; y
```

Output:

```
UShort Range = 0 to 65535
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__Ushort`.

Differences from QB

- New to FreeBASIC

See also

- [Short](#)

- CUShort

Using (Namespaces)



Brings namespace symbols into the current scope

Syntax

```
Using identifier [, identifier [, ...] ]
```

Parameters

identifier: The name of the **Namespace** that you want to use.

Description

The **using** command allows all symbols from a given namespace to be accessed without the namespace's name prefix. Unlike C++ but like C **Namespace** keyword is not needed after **using**, because individual symbols cannot be inherited from a namespace. Inheriting a whole namespace save typing, but sometimes some meaning of the code can be lost, and conflicts with other symbols could be created.

Example

```
Namespace Sample
    Type T
        x As Integer
    End Type
End Namespace

'' Just using the name T would not find the symbol
'' because it is inside a namespace.
Dim SomeVariable As Sample.T

'' Now the whole namespace has been inherited into
'' the global namespace.
Using Sample

'' This statement is valid now, since T exists
```

```
' ' without the "Sample." prefix.  
Dim OtherVariable As T
```

Differences from QB

- QB had the `using` keyword, but for other purposes. Namespaces exist in QB.

See also

- [\(Print | ?\) Using](#)
- [Palette Using](#)
- [Namespace](#)

Returns the current argument from a variable argument list.

Syntax

```
variable = va_arg ( argument_list, datatype )
```

Description

The **va_arg** macro allows the use of a variable number of arguments within a function. **va_arg** returns the current argument in the list, *argument_list*, with an expected data type of *datatype*. Before **va_arg** can be used, it must be initialized with the command **va_first**. Unlike the C macro with the same name, **va_arg** does not automatically increment *argument_list* to the next argument within the list. Instead **va_next** must be used to find the next argument.

Example

See the **va_First()** examples.

Dialect Differences

- Not available in the **-lang qb** dialect unless referenced with the alias **__va_arg**.

Differences from QB

- New to FreeBASIC

See also

- **...** (Ellipsis)
- **va_first**
- **va_next**

Returns a pointer to the first argument in a variable argument list

Syntax

```
pointer_variable = va_first()
```

Description

The `va_first` function provides an untyped **pointer** value that points to

Example

```
Function average cdecl(count As Integer, ... ) As  
    Dim arg As Any Ptr  
    Dim sum As Double = 0  
    Dim i As Integer  
  
    arg = va_first()  
  
    For i = 1 To count  
        sum += va_arg(arg, Double)  
        arg = va_next(arg, Double)  
    Next  
  
    Return sum / count  
End Function  
  
Print average(4, 3.4, 5.0, 3.2, 4.1)  
Print average(2, 65.2, 454.65481)  
Sleep
```

The output would look like:

```
3.925  
259.927405
```

```

'' Example of a simple custom printf
Sub myprintf cdecl(ByRef formatstring As String, .
    '' Get the pointer to the first var-arg
    Dim As Any Ptr arg = va_first()

    '' For each char in format string...
    Dim As UByte Ptr p = StrPtr(formatstring)
    Dim As Integer todo = Len(formatstring)
    While (todo > 0)
        Dim As Integer char = *p
        p += 1
        todo -= 1

        '' Is it a format char?
        If (char = Asc("%")) Then
            If (todo = 0) Then
                '' % at the end
                Print "%";
                Exit While
            End If

            '' The next char should tell the type
            char = *p
            p += 1
            todo -= 1

            '' Print var-arg, depending on the type
            Select Case char
                '' integer?
                Case Asc("i")
                    Print Str(va_arg(arg, Integer));
                    '' Note, different from C: va_next
                    '' used as va_arg() won't update t
                    arg = va_next(arg, Integer)

                '' long integer? (64-bit)

```

```

    Case Asc("l")
        Print Str(va_arg(arg, LongInt));
        arg = va_next(arg, LongInt)

    ' single or double?
    ' Note: because the C ABI, all single
    ' var-args are converted to doubles.
    Case Asc( "f" ), Asc( "d" )
        Print Str(va_arg(arg, Double));
        arg = va_next(arg, Double)

    ' string?
    Case Asc("s")
        ' Strings are passed byval, so th
        Print *va_arg(arg, ZString Ptr);
        arg = va_next(arg, ZString Ptr)

    End Select

    ' Ordinary char, just print as-is
Else
    Print Chr( char );
End If
Wend
End Sub

Dim As String s = "bar"

myprintf(!"integer=%i, longint=%l single=%f, c
        1, 111 Shl 32, 2.2, 3.3, "foo", s)

Sleep

```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- [... \(Ellipsis\)](#)
- [va_arg](#)
- [va_next](#)

Returns a pointer to the next argument in a variable argument list

Syntax

```
Argument_Pointer = va_next ( Argument_List, datatype )
```

Description

The `va_next` macro points to the next argument within the list `Argument_List`, `datatype` being the type of the current argument being stepped over.

Example

See the `va_First()` examples.

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__va_next`.

Differences from QB

- New to FreeBASIC

See also

- `... (Ellipsis)`
- `va_arg`
- `va_first`

Converts a string to a floating point number

Syntax

```
Declare Function Val ( ByRef str As Const String ) As Double  
Declare Function Val ( ByRef str As Const WString ) As Double
```

Usage

```
result = Val( strnum )
```

Parameters

strnum
the string containing a number to convert

Return Value

Returns a converted **Double** precision number

If the first character of the string is invalid, **val** will return 0.

Description

val("10") will return 10.0, and **val**("10.10") will return 10.1. The function parses the string from the left, skipping any white space, and returns the longest number it can read, stopping at the first non-suitable character it finds. Scientific notation is recognized, with "D" or "E" used to specify the exponent.

val can be used to convert integer numbers in binary / octal / hexadecimal format, if they have the relevant identifier ("&B;" / "&O;" / "&H;") prefixed, for example: **val**("&HFF;") returns 255.

Note:

If you want to get an integer value from a string, consider using **ValInt** or **ValLng** instead. They are faster, since they don't use floating-point numbers, and only **ValLng** provides full 64-bit precision for **LongInt**

types.

If you want to convert a number into string format, use the `str` function

Example

```
Dim a As String, b As Double
a = "2.1E+30xa211"
b = Val(a)
Print a, b
```

```
2.1E+30xa211  2.1e+030
```

Differences from QB

- None

See also

- `Cdbl`
- `ValInt`
- `ValUInt`
- `ValLng`
- `ValULng`
- `Str`
- `Chr`
- `Asc`

Converts a string to a 64bit integer

Syntax

```
Declare Function ValLng ( ByRef strnum As Const String ) As  
LongInt  
Declare Function ValLng ( ByRef strnum As Const WString ) As  
LongInt
```

Usage

```
result = ValLng ( strnum )
```

Parameters

strnum
the string to convert

Return Value

Returns a **LongInt** of the converted string

If the first character of the string is invalid, **valLng** will return 0.

Description

For example, **valLng**("10") will return 10, and **valLng**("10.60") will return 10 as well. The function parses the string from the left, skipping any white space, and returns the longest number it can read, stopping at the first non-suitable character it finds. Any non-numeric characters including decimal points and exponent specifiers, are considered non-suitable, for example, **valLng**("23.1E+6") will just return 23.

valLng can be used to convert integer numbers in **Binary / Octal / Hexadecimal** format, if they have the relevant identifier ("**&B**;" / "**&O**;" / "**&H**;" prefixed, for example: **valLng**("&HFF;") returns 255.

If you want to convert a number into string format, use the **str** function

Example

```
Dim a As String, b As LongInt
a = "20xa211"
b = ValLng(a)
Print a, b
```

```
20xa211 20
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__ValLng`.

Differences from QB

- New to FreeBASIC

See also

- `CLngInt`
- `Val`
- `ValInt`
- `ValULng`
- `Str`
- `Chr`
- `Asc`

Converts a string to a 32bit integer

Syntax

```
Declare Function ValInt ( ByRef strnum As Const String ) As Long
Declare Function ValInt ( ByRef strnum As Const WString ) As Lon
```

Usage

```
result = ValInt ( strnum )
```

Parameters

strnum
the string to convert

Return Value

Returns a **Long** value of the converted string

If the first character of the string is invalid, **valInt** will return 0.

Description

For example, **valInt**("10") will return 10, and **valInt**("10.60") will return 10 as well. The function parses the string from the left, skipping any white space, and returns the longest number it can read, stopping at the first non-suitable character it finds. Any non-numeric characters including decimal points and exponent specifiers, are considered non-suitable, for example, **valInt**("23.1E+6") will just return 23.

valInt can be used to convert integer numbers in **Binary / Octal / Hexadecimal** format, if they have the relevant identifier ("&B;" / "&O;" / "&H;") prefixed, for example: **valInt**("&HFF;") returns 255.

If you want to convert a number into string format, use the **str** function

Example

```
Dim a As String, b As Integer
a = "20xa211"
b = ValInt(a)
Print a, b
```

```
20xa211  20
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__ValInt`.

Differences from QB

- New to FreeBASIC

See also

- `CLng`
- `Val`
- `ValUInt`
- `ValLng`
- `Str`
- `Chr`
- `Asc`

Converts a string to an unsigned 32bit integer

Syntax

```
Declare Function ValUInt ( ByRef strnum As Const String ) As  
Ulong  
Declare Function ValUInt ( ByRef strnum As Const WString ) As  
Ulong
```

Usage

```
result = ValUInt ( strnum )
```

Parameters

strnum
the string to convert

Return Value

Returns a **ulong** value of the converted string

If the first character of the string is invalid, **ValUInt** will return 0.

Description

For example, **ValUInt**("10") will return 10, and **ValUInt**("10.60") will return 10 as well. The function parses the string from the left, skipping any white space, and returns the longest number it can read, stopping at the first non-suitable character it finds. Any non-numeric characters including decimal points and exponent specifiers, are considered non-suitable, for example, **ValUInt**("23.1E+6") will just return 23.

ValUInt can be used to convert integer numbers in **Binary / Octal / Hexadecimal** format, if they have the relevant identifier ("**&B**;" / "**&O**;" / "**&H**;" prefixed, for example: **ValUInt**("&HFF;") returns 255.

If you want to convert a number into string format, use the **str** function

Example

```
Dim a As String, b As UInteger
a = "20xa211"
b = ValUInt(a)
Print a, b
```

```
20xa211  20
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__Valuint`.

Differences from QB

- New to FreeBASIC

See also

- `Val`
- `ValInt`
- `ValULng`
- `CULng`
- `Str`
- `Chr`
- `Asc`

Converts a string to a unsigned 64bit integer

Syntax

```
Declare Function ValULng ( ByRef strnum As Const String ) As  
    ULongInt  
Declare Function ValULng ( ByRef strnum As Const WString ) As  
    ULongInt
```

Usage

```
result = ValULng ( strnum )
```

Parameters

strnum
the string to convert

Return Value

Returns a **ULongInt** of the converted string

If the first character of the string is invalid, **valULng** will return 0.

Description

For example, **valULng**("10") will return 10, and **valULng**("10.60") will return 10 as well. The function parses the string from the left, skipping any white space, and returns the longest number it can read, stopping at the first non-suitable character it finds. Any non-numeric characters including decimal points and exponent specifiers, are considered non-suitable, for example, **valULng**("23.1E+6") will just return 23.

valULng can be used to convert integer numbers in **Binary / Octal / Hexadecimal** format, if they have the relevant identifier ("**&B**;" / "**&O**;" / "**&H**;" prefixed, for example: **valULng**("&HFF;") returns 255.

If you want to convert a number into string format, use the **str** function

Example

```
Dim a As String, b As ULongInt
a = "20xa211"
b = ValULng(a)
Print a, b
```

```
20xa211  20
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__ValuIng`.

Differences from QB

- New to FreeBASIC

See also

- `CULngInt`
- `Val`
- `ValUInt`
- `ValLNg`
- `Str`
- `Chr`
- `Asc`

Declares a variable whose type is implied from the initializer expression

Syntax

```
Var [Shared] symbolName = expression[, symbolName = expression]
```

Description

`var` declares a variable whose type is implied from the initializer expression. It is illegal to specify an explicit type in a `var` declaration. The initializer expression can be either a constant or any variable of any type.

Note: `wString` is not supported with `var`, due to the fact that there is no var-len `wString` type. This isn't likely to change, due to the complexities involved with handling Unicode.

Since the type of the variable is inferred from what you assign into it, it's helpful to know how literals work. Any literal number without a decimal point defaults to `Integer`. A literal number *with* a decimal point defaults to `Double`. See `ProPgLiterals` for further information.

All `zString` expressions, including string literals and dereferenced `zString Ptrs`, will be given the `String` variable type.

Explicit suffixes may be used on literal variables, to change/clarify the type. See `Literals` and `Variable Types` for some more information about suffixes that can be used on literals.

Note: Suffixes must appear on the initializer, not on the variable. Trying to use `var` with a variable that has a suffix will throw a compile error.

Example

```
Var a = Cast(Byte, 0)
Var b = Cast(Short, 0)
```

```

Var c = Cast(Integer, 0)
Var d = Cast(LongInt, 0)
Var au = Cast(UByte, 0)
Var bu = Cast(UShort, 0)
Var cu = Cast(UInteger, 0)
Var du = Cast(ULongInt, 0)
Var e = Cast(Single, 0.0)
Var f = Cast(Double, 0.0)
Var g = @c      '' integer ptr
Var h = @a      '' byte ptr
Var s2 = "hello" '' var-len string

Var ii = 6728   '' implicit integer
Var id = 6728.0 '' implicit double

Print "Byte: ";Len(a)
Print "Short: ";Len(b)
Print "Integer: ";Len(c)
Print "Longint: ";Len(d)
Print "UByte: ";Len(au)
Print "UShort: ";Len(bu)
Print "UInteger: ";Len(cu)
Print "ULongint: ";Len(du)
Print "Single: ";Len(e)
Print "Double: ";Len(f)
Print "Integer Pointer: ";Len(g)
Print "Byte Pointer: ";Len(h)
Print "Variable String: ";Len(s2)
Print
Print "Integer: ";Len(ii)
Print "Double: ";Len(id)

Sleep

```

Differences from QB

- New to FreeBASIC 0.17

Dialect Differences

- Only valid in the *-lang fb* dialect.

See also

- **Common**
- **Dim**
- **Erase**
- **Extern**
- **LBound**
- **ReDim**
- **Preserve**
- **Shared**
- **Static**
- **UBound**

Operator VarPtr (Variable Pointer)



Returns the address of a variable or object

Syntax

```
Declare Operator VarPtr ( ByRef lhs As T ) As T Ptr
```

Syntax

```
result = VarPtr ( lhs )
```

Parameters

lhs

A variable or object.

T

Any data type.

Return Value

Returns the address of a variable or object.

Description

This operator returns the address of its operand.

When the operand is of type **String**, the address of the internal string **StrPtr (String Pointer)** to retrieve the address of the string data.

The operand cannot be an array, but may be an array element. For example, the address of "myarray(0)".

Example

```
Dim a As Integer, addr As Integer
a = 10

'' place the address of a in addr
addr = CInt( VarPtr(a) )
```

```
' ' change all 4 bytes (size of INTEGER) of a
Poke Integer, addr, -1000
Print a

' ' place the address of a in addr (same as above)
addr = CInt( @a )

' ' print the least or most significant byte, deper
Print Peek( addr )
```

Differences from QB

- None

See also

- **Pointers**
- **Peek**
- **Poke**

View Print



Sets the printable area of the screen

Syntax

```
View Print [ firstrow To lastrow ]
```

Parameters

firstrow

first row of print area

lastrow

last row of print area

Description

Sets the boundaries of the console screen text area to the lines starting including last. Lines are counted starting with 1. The text cursor is moved to the beginning of the first line specified.

If the row numbers are omitted, the entire screen is used as the text area.

Example

```
Cls  
View Print 5 To 6  
Color , 1  
' ' clear only View Print area  
Cls
```

View Print can be used in graphics mode to avoid the text output overwriting the graphics.

```
Screen 12  
Dim As Integer R,Y,x,y1  
Dim As Single y2  
View Print 20 To 27  
Line (0,0)-(639,300),1,BF
```

```
Line (100,50)-(540,200),0,BF
Do
  r = (r + 1) And 15
  For y = 1 To 99
    y1 = ((1190 \ y + r) And 15)
    y2 = 6 / y
    For x = 100 To 540
      PSet (x, y + 100), CInt((319 - x) * y2) And 15
    Next x,y
  If r=0 Then Color Int(Rnd*16): Print "blah"
Loop Until Len(Inkey)
```

Differences from QB

- None.

See also

- Cls
- (Print | ?)
- Color

Sets new physical coordinate mapping and clipping region

Syntax

View

```
View ( x1, y1 )-( x2, y2 ) [ [ , fill_color ] [ , border_color ] ]
```

```
View Screen ( x1, y1 )-( x2, y2 ) [ [ , fill_color ] [ ,  
border_color ] ]
```

Parameters

x1 **As Integer**, *y1* **As Integer**

The horizontal and vertical offsets, in pixels, of one corner of the viewport relative to the top-left corner of the screen.

x2 **As Integer**, *y2* **As Integer**

The horizontal and vertical offsets, in pixels, of the opposite corner of the viewport relative to the top-left corner of the screen.

fill_color **As UInteger**

The color to fill the new viewport.

border_color **As UInteger**

The color of the border to draw around the new viewport.

Description

The *viewport*, or clipping region, is a rectangular area of the graphics screen, outside of which no drawing will be done. That is, only drawing done within this area will be shown. A graphics screen must be created with **Screen** or **ScreenRes** before calling **View** or **View Screen**.

The first statement sets the viewport to encompass the entire screen, which is the default viewport for a new graphics screen.

The second and third statements both allow a new viewport to be defined. The corners of the viewport are specified by the *x1*, *y1*, *x2* and *y2* parameters. *fill_color* and *border_color* are both in the format accepted by **color**. The indicated effects for each parameter only occur if that parameter is specified.

The second statement modifies the coordinate mapping of the graphic screen such that coordinates specified for drawing statements and procedures are relative to the top-left corner of the viewport.

The third statement modifies the coordinate mapping of the graphics screen such that coordinates specified for drawing statements and procedures are relative to the top-left corner of the screen.

Example

```
Screen 12
Dim ip As Any Ptr
Dim As Integer x, y

'simple sprite
ip = ImageCreate(64,64)
For y = 0 To 63
  For x = 0 To 63
    PSet ip, (x, y), (x\4) Xor (y\4)
  Next x
Next y

'viewport with blue border
Line (215,135)-(425,345), 1, bf
View (220,140)-(420,340)

'move sprite around the viewport
Do

  x = 100*Sin(Timer*2.0)+50
  y = 100*Sin(Timer*2.7)+50

  ScreenSync
  ScreenLock

  'clear viewport and put image
  Cls 1
  Put (x, y), ip, PSet
```

```
ScreenUnlock
```

```
Loop While Inkey = ""
```

```
ImageDestroy(ip)
```

Differences from QB

- QBASIC preserves the `WINDOW` coordinate mapping after subsequent calls to `VIEW`.
- FreeBASIC's current behavior is to preserve the `WINDOW` coordinates after calls to `VIEW`, or when working on images, meaning that the coordinate mapping may undergo scaling/translations if the viewport changes. (If a `WINDOW` hasn't been set, there is no coordinate mapping, and so it doesn't change after calls to `VIEW`.) The behavior may change in future but consistent behavior can be assured over inconsistent viewport coordinates by re-calling `WINDOW` whenever you change the `VIEW`.

See also

- [View Print](#)
- [Screen \(Graphics\)](#)
- [Window](#)
- [PMap](#)

Declare virtual methods

Syntax

```
Type typename Extends base_typename  
Declare Virtual Sub|Function|Property|Operator|Destructor ...  
End Type
```

Description

Virtual methods are methods that can be overridden by data types derived from the type they were declared in, allowing for polymorphism. In contrast to **Abstract** methods, virtual methods must have an implementation, which is used when the virtual is not overridden.

A derived type can override virtual methods declared in its base type by declaring a method with the same identifier and signature, meaning same number and type of parameters, same return type (if any) and same calling convention:

- if that differs only in parameter passing mode or calling convention or return type, then an overriding error is returned at compile time,
- otherwise shadowing only is permitted for any other signature difference, corresponding to case where both methods would be overloadable.

The property of being a virtual method is not implicitly inherited by the overriding method in the derived type.

When calling virtual methods, the compiler may need to do a vtable lookup in order to find out which method must be called for a given object. This requires an extra hidden vtable pointer field to be added at the top of each type with virtual methods. This hidden vptr is provided by the built-in **Object** type. Because of that, virtual methods can only be declared in a type that directly or indirectly **Extends Object**.

Constructors cannot be virtual because they create objects, while

virtual methods require an already-existing object with a specific type. The type of the constructor to call is determined at compile-time from the code.

In addition, when calling a virtual method inside a constructor, only the version of the method corresponding to an object of type of this constructor is used. That is because the vptr has not yet been set up by the derived type constructor, but only by the local type constructor.

Destructors often must be virtual when deleting an object manipulate through a pointer to its base type, so that the destruction starts at the most derived type and works its way down to the base type. To do this it may be necessary to add virtual destructors with an empty body anywhere an explicit destruction was not yet required, in order to supersede each non-virtual implicit destructor induced by the destructor in its base.

On the other hand, when calling a virtual (or abstract) method inside a destructor (virtual or not), only the version of the method corresponding to an object of type of this destructor is used because the vptr is reset at the top of the destructor according to its own type's vtable. This avoids to access child methods and so to refer to child members previously destroyed by the child destructor execution.

For member methods with `virtual` in their declaration, `virtual` can also be specified on the corresponding method bodies, for improved code readability.

Note: In a multi-level inheritance, a same named method (same identifier and signature) can be declared **Abstract**, **virtual** or normal (without specifier) at each inheritance hierarchy level. When there is mixing of specifiers, the usual order is abstract -> virtual -> normal, from top to bottom of the inheritance hierarchy.

The access control (**Public/Protected/Private**) of an overriding method is not taken into account by the internal polymorphism process, but only for the initial call at compile-time.

Base.method() calls always the base's own method, never the overriding method.

A derived static method cannot override a base virtual/abstract method, but can shadow any base method (including virtual/abstract).

Example

```
Type Hello extends object
  Declare virtual Sub hi( )
End Type

Type HelloEnglish extends Hello
  Declare Sub hi( )
End Type

Type HelloFrench extends Hello
  Declare Sub hi( )
End Type

Type HelloGerman extends Hello
  Declare Sub hi( )
End Type

Sub Hello.hi( )
  Print "hi!"
End Sub

Sub HelloEnglish.hi( )
  Print "hello!"
End Sub

Sub HelloFrench.hi( )
  Print "Salut!"
End Sub

Sub HelloGerman.hi( )
  Print "Hallo!"
End Sub

Randomize( Timer( ) )
```

```
Dim As Hello Ptr h

For i As Integer = 0 To 9
  Select Case( Int( Rnd( ) * 4 ) + 1 )
  Case 1
    h = New HelloEnglish
  Case 2
    h = New HelloFrench
  Case 3
    h = New HelloGerman
  Case Else
    h = New Hello
  End Select

  h->hi( )
  Delete h
Next
```

Dialect Differences

- Only available in the *-lang fb* dialect.

Differences from QB

- New to FreeBASIC

See also

- [Type](#)
- [Object](#)
- [Extends](#)
- [Abstract](#)

Wait



Reads from a hardware port with a mask.

Syntax

```
Declare Function Wait ( ByVal port As UShort, ByVal and_mask As Long = 0 ) As Long
```

Usage

```
Wait port, and_value [, xor_value]
```

Parameters

port

Port to read.

and_mask

Mask value to **And** the port value with.

xor_mask

Mask value to **Xor** the port value with.

Return Value

0 if successful, -1 on failure.

Description

`wait` keeps reading *port* until the reading ANDed with *and_mask* and or *xor_mask* gives a non-zero result.

Example

```
Wait &h3da, &h8 'Old Qbasic way of waiting for the  
ScreenSync 'FreeBASIC way of accomplishing the same
```

Platform Differences

- In the Windows and Linux versions three port numbers (&H3C) hooked by the graphics library when a graphics mode is in use handling as in QB. This use is deprecated; use **Palette** to retrieve
- Using true port access in the Windows version requires the process for the present session. For that reason, Windows executables should be run with administrator permissions each time the computer don't require admin rights as they just use the already installed in size and is embedded in the executable.

See also

- **Inp**
- **Out**

Returns the binary **wString** (Unicode) representation of a number

Syntax

```
Declare Function WBin ( ByVal number As UByte ) As WString
Declare Function WBin ( ByVal number As UShort ) As WString
Declare Function WBin ( ByVal number As ULong ) As WString
Declare Function WBin ( ByVal number As ULongInt ) As WString
Declare Function WBin ( ByVal number As Const Any Ptr ) As
WString
```

```
Declare Function WBin ( ByVal number As UByte, ByVal digits As
Long ) As WString
Declare Function WBin ( ByVal number As UShort, ByVal digits As
Long ) As WString
Declare Function WBin ( ByVal number As ULong, ByVal digits As
Long ) As WString
Declare Function WBin ( ByVal number As ULongInt, ByVal digits A
Long ) As WString
Declare Function WBin ( ByVal number As Const Any Ptr, ByVal
digits As Long ) As WString
```

Usage

```
result = WBin( number [, digits] )
```

Parameters

number

A whole number or expression evaluating to a whole number.

digits

Optional number of digits to return.

Return Value

Returns a binary **wString** representation of *number*, truncated or padded with zeros ("0") to fit the number of digits, if specified.

Description

Returns a **wString** (Unicode) representing the binary value of the integer *number*. Binary digits range from 0 to 1.

If you specify *digits* > 0, the result wstring will be exactly that length. will be truncated or padded with zeros on the left, if necessary.

The length of the returned string will not be longer than the maximum number of digits required for the type of *expression* (32 for a **Long**, 64 for floating point or **LongInt**)

Example

```
Print WBin(54321)
Print WBin(54321, 5)
Print WBin(54321, 20)
```

will produce the output:

```
1101010000110001
10001
00001101010000110001
```

Platform Differences

- Unicode strings are not supported in the DOS port of FreeBASIC.

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__wbin`.

Differences from QB

- New to FreeBASIC

See also

- **Bin**
- **WHex**
- **WOct**

Returns a wide-character string containing one or more Unicode characters.

Syntax

```
Declare Function WChr ( ByVal ch As Integer [, ... ] ) As WString
```

Usage

```
result = WChr( ch0 [, ch1 ... chN ] )
```

Parameters

ch

The Unicode integer value of a character.

Return Value

Returns a wide-character string.

Description

`wchr` returns a wide-character string containing the character(s) represented by the integer value(s).

When `wchr` is used with numerical constants or literals, the result is evaluated at compile time and can be used in variable initializers.

Not all Unicode characters can be displayed on any machine, the characters currently in use in the console. Graphics modes can't display Unicode characters that are not supported by the console.

Example

```
Print "The character represented by the UNICODE code point 933 is: "; WChr(933)
Print "Multiple UNICODE characters: "; WChr(933, 934, 935)
```

will produce the output:

The character represented by the UNICODE code of 934 is: Φ
Multiple UNICODE characters: YΦX

Platform Differences

- DOS does not support `wchr`.

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- `Chr`
- `WStr`

Weekday



Gets the number of day of the week from a **Date Serial**

Syntax

```
Declare Function Weekday ( ByVal serial As Double , ByVal firstd  
fbusesystem ) As Long
```

Usage

```
#include "vbcompat.bi"  
result = Weekday( date_serial [, firstdayofweek ] )
```

Parameters

date_serial
the date
firstdayofweek
the first day of the week

Return Value

Returns the week day number from a variable containing a date in **Da**

Description

The week day values must be in the range 1-7, its meaning depends on the *firstdayofweek* parameter

firstdayofweek is optional.

value	first day of week	constant
omitted	sunday	
0	local settings	fbUseSystem
1	sunday	fbSunday
2	monday	fbMonday
3	tuesday	fbTuesday
4	wednesday	fbWednesday

5	thursday	fbThursday
6	friday	fbFriday
7	saturday	fbSaturday

The compiler will not recognize this function unless `vbcompat.bi` is inc

Example

```
#include "vbcompat.bi"

Dim a As Double = DateSerial (2005, 11, 28) + Time

Print Format(a, "yyyy/mm/dd hh:mm:ss "); Weekday(a
```

Differences from QB

- Did not exist in QB. This function appeared in PDS and VBDO:

See also

- [Date Serials](#)

WeekdayName



Gets the name of a week day from its integral representation

Syntax

```
Declare Function WeekdayName ( ByVal weekday As , ByVal abbrevia  
firstdayofweek As Long = fbUseSystem ) As String
```

Usage

```
#include "vbcompat.bi"  
result = WeekdayName( weekday [, abbreviate [, firstdayofweek ]
```

Parameters

weekday
the number of the day of the week

abbreviate
flag to indicate that name should be abbreviated

firstdayofweek
first day of the week

Return Value

Returns the local operating system language day of week name from

Description

How *weekday* is interpreted depends on the *firstdayofweek* parameter

If *abbreviate* is true, a 3 letter abbreviation is returned, if false or omit returned.

firstdayofweek is an optional parameter specified as follows:

value	first day of week	constant
omitted	sunday	
0	local settings	fbUseSystem
1	sunday	fbSunday

2	monday	fbMonday
3	tuesday	fbTuesday
4	wednesday	fbWednesday
5	thursday	fbThursday
6	friday	fbFriday
7	saturday	fbSaturday

The compiler will not recognize this function unless `vbcompat.bi` or `date`

Example

```
#include "vbcompat.bi"

Dim a As Double = DateSerial(2005, 11, 28) + TimeSerial(12, 34, 56)
Print Format(a, "yyyy/mm/dd hh:mm:ss "); WeekdayName(Weekday(a))
```

Differences from QB

- Did not exist in QB. This function appeared in Visual Basic.

See also

- [Date Serials](#)

Control flow statement.

Syntax

```
while [condition]  
  [statement block]  
wend
```

Description

wend specifies the end of a `while...wend` loop block.

Differences from QB

- None

See also

- `While...Wend`

Control flow statement.

Syntax

```
Do while condition
[statement block]
Loop
or
Do
[statement block]
Loop while condition
or
while [condition]
[statement block]
Wend
```

Description

`while` specifies that a loop block will continue if the *condition* following it evaluates as true. This *condition* is checked during each loop iteration.

Differences from QB

- None

See also

- [Do...Loop](#)
- [While...Wend](#)

While...Wend



Control flow statement for looping

Syntax

```
While [condition]
[statement block]
Wend
```

Description

The **while** statement will cause the following set of statements in the *statement block* to be executed repeatedly while the expression *condition* evaluates to true.

If *condition* evaluates to false when the **while** statement is first executed, execution resumes immediately following the enclosing **wend** statement.

If an **Exit while** statement is encountered inside the *statement block*, execution resumes immediately following the enclosing **wend** statement. If a **Continue while** statement is encountered inside the *statement block*, the remainder of the *statement block* is skipped and execution resumes at the **while** statement.

Like all control flow statements, the **while** statement can be nested, that is, it can contain another **while** statement.

note: the while keyword is also used in the Do...Loop statement to indicate that the do statement becomes functionally equivalent to the while statement. In this case, the Loop and wend, respectively.

Example

In this example, a **while** loop is used to reverse a string by iterating through the string from the last character to the first (0 being the first index in the string).

```
Dim As String sentence
sentence = "The quick brown fox jumps over the lazy dog"

Dim As String ecnetnes
Dim As Integer index
index = Len( sentence ) - 1
```

```

While( index >= 0 )
    ecnetnes += Chr( sentence[index] )
    index -= 1
Wend

Print "original: " ; sentence ; ""
Print "reversed: " ; ecnetnes ; ""

End 0

```

Dialect Differences

- In the *-lang qb* and *-lang fblite* dialects, variables declared inside a block are **scope** as in QB
- In the *-lang fb* and *-lang deprecated* dialects, variables declared inside the block, and can't be accessed outside it.

Differences from QB

- None

See also

- **Exit**
- **Continue**
- **Do...Loop**

Returns the hexadecimal **wstring** (Unicode) representation of a number

Syntax

```
Declare Function WHex ( ByVal number As UByte ) As WString
Declare Function WHex ( ByVal number As UShort ) As WString
Declare Function WHex ( ByVal number As ULong ) As WString
Declare Function WHex ( ByVal number As ULongInt ) As WString
Declare Function WHex ( ByVal number As Const Any Ptr ) As
WString
```

```
Declare Function WHex ( ByVal number As UByte, ByVal digits As
Long ) As WString
Declare Function WHex ( ByVal number As UShort, ByVal digits As
Long ) As WString
Declare Function WHex ( ByVal number As ULong, ByVal digits As
Long ) As WString
Declare Function WHex ( ByVal number As ULongInt, ByVal digits A
Long ) As WString
Declare Function WHex ( ByVal number As Const Any Ptr, ByVal
digits As Long ) As WString
```

Usage

```
result = WHex( number [, digits ] )
```

Parameters

number

A whole number or expression evaluating to a whole number.

digits

Optional number of digits to return.

Return Value

Returns a hexadecimal **wstring** representation of *number*, truncated or padded with zeros ("0") to fit the number of digits, if specified.

Description

Hexadecimal digits range from 0-9, or A-F.

If you specify *digits* > 0, the resulting **wstring** will be exactly that length. It will be truncated or padded with zeros on the left, if necessary.

The length of the wstring will not go longer than the maximum number of digits required for the type of *expression* (8 for a **Long**, 16 for floating point or **LongInt**)

Example

```
Print Hex(54321)
Print Hex(54321, 2)
Print Hex(54321, 5)
```

will produce the output:

```
D431
31
0D431
```

Platform Differences

- Unicode strings are not supported in the DOS port of FreeBASIC.

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__whex`.

Differences from QB

- New to FreeBASIC.

See also

- [Hex](#)
- [WBin](#)
- [WOct](#)

Sets or gets the number of rows and columns of the display

Syntax

```
width [columns] [, rows]  
width LPrint columns  
width { #filenum | devicename }, columns  
result = width( )
```

Parameters

columns
columns (in characters) for output
rows
rows (in characters) for output
filenum
file number to apply to
devicename
device name to apply to

Return Value

Returns a 32 bit **Long** where the **High Word** is the number of rows and the **Low Word** is the number of columns currently set.

Description

Sets the maximum number of columns of characters of an output device (console, printer or text file). If text sent to the device reaches the width an automatic carriage return is generated.

Using **width** as a function returns the current console width in the low word and the current height in the high word.

If a device is not given then **width** takes effect on the active console/graphics screen, and a second argument specifying maximum number of rows is allowed.

In graphics modes **width** is used to indirectly select the font size by

setting one of the character height * width pairs allowed (See [Screen \(Graphics\)](#)). If *rows / cols* is an invalid combination, no changes are made to the screen display.

Valid font heights are 8 pixels, 14 pixels and 16 pixels. The fonts all have a fixed width of 8 pixels.

Using the `width` command in graphic mode also forces a screen clear (`cls`).

Example

```
Dim As Integer w
w = Width
Print "rows: " & HiWord(w)
Print "cols: " & LoWord(w)
```

```
'Set up a graphics screen
Const W = 320, H = 200
ScreenRes W, H

Dim As Integer twid, tw, th

' Fetch and print current text width/height:
twid = Width()
tw = LoWord(twid): th = HiWord(twid)
Print "Default for current screen (8*8)"
Print "Width: " & tw
Print "Height: " & th
Sleep

Width W\8, H\16 ' Use 8*16 font

twid = Width()
tw = LoWord(twid): th = HiWord(twid)
```

```

Print "Set to 8*16 font"
Print "Width: " & tw
Print "Height: " & th
Sleep

Width W\8, H\14 ' ' Use 8*14 font

twid = Width()
tw = LoWord(twid): th = HiWord(twid)
Print "Set to 8*14 font"
Print "Width: " & tw
Print "Height: " & th
Sleep

Width W\8, H\8 ' ' Use 8*8 font

twid = Width()
tw = LoWord(twid): th = HiWord(twid)
Print "Set to 8*8 font"
Print "Width: " & tw
Print "Height: " & th
Sleep

```

Platform Differences

- In a Windows console any values > 0 can be used in windowed mode.
- On a DOS or Windows full-screen console, the valid dimensions depend on the capabilities of the hardware.
- Linux doesn't allow applications to change the console size.

Differences from QB

- *columns* was limited to 40 or 80, while *rows* could be 25, 30, 43, 50 or 60, depending on the graphics hardware and screen mode being used.

See also

- **LoWord**
- **HiWord**
- **CsrLin**
- **Pos**

Window



Sets new view coordinates mapping for current viewport

Syntax

```
Window [ [Screen] ( x1, y1 )-( x2, y2 ) ]
```

Parameters

Screen

Optional argument specifying y coordinates increase from top to bottom
(x1, y1)-(x2, y2)

New floating point values corresponding to the opposite corners of the

Description

`Window` is used to define a new coordinates system. (x1, y1) and (x2, y2) are affected by this new mapping. If `Screen` is omitted, the new coordinate system is defined relative to the screen.

FreeBASIC's current behavior is to keep track of the corners of the window. The `Window` corners are also currently taken into account when working with the `Screen` parameter.

When there is no `Window` in effect, there is no coordinate mapping in effect.

Example

```
' ' The program shows how changing the view coordinates
' ' The effect is one of zooming in and out:
' '   - As the viewport coordinates get smaller, the
' '   - As the viewport coordinates get larger, the
' '   - As the viewport coordinates get larger, the

Declare Sub Zoom (ByVal X As Integer)
Dim As Integer X = 500, Xdelta = 50

Screen 12
Do
  Do While X < 525 And X > 50
    X += Xdelta
  ' ' Change window coordinates
```

```

Zoom(X)
If Inkey <> "" Then Exit Do, Do '' Stop if ke
Sleep 100
Loop
X -= Xdelta
Xdelta *= -1 '' Reverse si
Loop

Sub Zoom (ByVal X As Integer)
Window (-X, -X)-(X, X) '' Define new
ScreenLock
Cls
Circle (0,0), 60, 11, , , 0.5, F '' Draw ellip
ScreenUnlock
End Sub

```

Screen 13

```

'' define clipping area
View ( 10, 10 ) - ( 310, 150 ), 1, 15

'' set view coordinates
Window ( -1, -1 ) - ( 1, 1 )

'' Draw X axis
Line (-1,0)-(1,0),7
Draw String ( 0.8, -0.1 ), "X"

'' Draw Y axis
Line (0,-1)-(0,1),7
Draw String ( 0.1, 0.8 ), "Y"

Dim As Single x, y, s

'' compute step size
s = 2 / PMap( 1, 0 )

```

```

'' plot the function
For x = -1 To 1 Step s
  y = x ^ 3
  PSet( x, y ), 14
Next x

'' revert to screen coordinates
Window

'' remove the clipping area
View

'' draw title
Draw String ( 120, 160 ), "Y = X ^ 3"

Sleep

```

Differences from QB

- QBASIC preserves the coordinate mapping after subsequent c
- FreeBASIC's current behavior is to preserve the WINDOW coc there is no coordinate mapping, and so it doesn't change after you change the VIEW.

See also

- **Screen (Graphics)**
- **View (Graphics)**
- **PMap**

WindowTitle



Sets the program window title

Syntax

```
Declare Sub WindowTitle ( ByRef title As Const String )
```

Usage

```
WindowTitle title
```

Parameters

title

the string to be assigned as new window title.

Description

This statement is useful to change the program window title. The new title set will become active immediately if the program already runs in windowed mode, otherwise will become the new title for any window produced by subsequent calls to the **Screen (Graphics)** statement. If this function is not called before setting a new windowed mode via **Screen (Graphics)**, the program window will use the executable file name (without the extension) as title by default.

This command has no effect in consoles.

Example

```
'Set screen mode
Screen 13

'Set the window title
WindowTitle "FreeBASIC example program"

Sleep
```

Platform Differences

- Not present in DOS version / target of FreeBASIC

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__Windowtitle`.

Differences from QB

- New to FreeBASIC

See also

- [Screen \(Graphics\)](#)

Winput()



Reads a number of wide-characters from console or file

Syntax

```
Declare Function WInput( ByVal num As Integer ) As WString  
Declare Function WInput( ByVal num As Integer, ByVal filenum As
```

Usage

```
result = WInput( num [, [#]filenum } )
```

Parameters

num
Number of characters to read.
filenum
File number of bound file or device.

Return Value

Returns a **WString** of the characters read.

Description

Reads a number of wide-characters from the console, or a bound file/

The first version waits for and reads *n* wide characters from the keyboard. If characters are not echoed to the screen.

The second version waits for and reads *n* wide characters from a file (

Note: FreeBASIC does not currently support reading wide-characters

Example

```
Dim char As WString * 2  
Dim filename As String, enc As String
```

```

Dim f As Integer

Line Input "Please enter a file name: ", filename
Line Input "Please enter an encoding type (optional): ", enc
If enc = "" Then enc = "ascii"

f = FreeFile
If Open(filename For Input Encoding enc As #f) = 0 Then

    Print "Press space to read a character from the file."

    Do

        Select Case Input(1)

            Case " " 'Space

                If EOF(f) Then

                    Print "You have reached the end of the file."
                    Exit Do

                End If

                char = WInput(1, f)
                Print char & " (char no " & Asc(char) & ") "

            Case Chr(27) 'Escape

                Exit Do

            Case Else

                End Select

        Loop

    Close #f

Else

```

```
Print "There was an error opening the file."  
End If
```

Dialect Differences

- Not available in the *-lang qb* dialect.

Differences from QB

- QB does not support Unicode

See also

- `Input()`
- `Open`

Statement block to allow implicit access to fields in a user defined type variable

Syntax

```
With user_defined_var
statements
End With
```

Description

The `with...End with` block allows the omission of the name of a variable or a user-defined **type** when referring to its fields. The fields may then be accessed with just a single period (.) before them, e.g. if the **type** contains a field element called "element", then it could be accessed within the block as ".element".

It can be used as a shorthand to save typing and avoid cluttering the source. `with` can also be used with dereferenced pointers, as the second example shows.

`with` blocks may be nested. In this case, only the innermost `with` block is active, and any outer ones are ignored until the inner one is closed again. See the third example for an illustration of this.

Internally, a reference to the variable is taken at the start of the `with` block and then is used to calculate any element accesses within the block. In other words, that this means that `goto` should not be used to jump into a `with` block otherwise the reference will not have been set, and the results of trying to access it will be undefined.

Note for with block used inside member procedure:

To access duplicated symbols defined outside the Type, use ". . someSymbol".

Example

```
Type rect_type
```

```

    x As Single
    y As Single
End Type

Dim the_rectangle As rect_type
Dim As Integer temp, t

With the_rectangle
    temp = .x
    .x = 234 * t + 48 + .y
    .y = 321 * t + 2
End With

```

```

Type rect_type
    x As Single
    y As Single
End Type

Dim the_rectangle As rect_type Ptr

the_rectangle = CAllocate( 5 * Len( rect_type ) )

Dim As Integer loopvar, temp, t

For loopvar = 0 To 4

    With the_rectangle[loopvar]

        temp = .x
        .x = 234 * t + 48 + .y
        .y = 321 * t + 2

    End With

Next

```

```

Type rect_type
  x As Single
  y As Single
End Type

Dim As rect_type rect1, rect2

'' Nested With blocks
With rect1

  .x = 1
  .y = 2

  With rect2

    .x = 3
    .y = 4

  End With

End With

Print rect1.x, rect1.y '' 1, 2
Print rect2.x, rect2.y '' 3, 4

```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the `__With`.

Differences from QB

- New to FreeBASIC

See also

- [Type](#)

Converts a number to a Unicode octal representation

Syntax

```
Declare Function WOct ( ByVal number As UByte ) As WString
Declare Function WOct ( ByVal number As UShort ) As WString
Declare Function WOct ( ByVal number As ULong ) As WString
Declare Function WOct ( ByVal number As ULongInt ) As WString
Declare Function WOct ( ByVal number As Const Any Ptr ) As
WString
```

```
Declare Function WOct ( ByVal number As UByte, ByVal digits As
Long ) As WString
Declare Function WOct ( ByVal number As UShort, ByVal digits As
Long ) As WString
Declare Function WOct ( ByVal number As ULong, ByVal digits As
Long ) As WString
Declare Function WOct ( ByVal number As ULongInt, ByVal digits A
Long ) As WString
Declare Function WOct ( ByVal number As Const Any Ptr, ByVal
digits As Long ) As WString
```

Usage

```
result = WOct( number [, digits ] )
```

Parameters

number

Number to convert to octal representation.

digits

Desired number of digits in the returned string.

Return Value

The Unicode octal representation of the number, truncated or padded with zeros ("0") to fit the number of digits, if specified.

Description

Returns the octal **wString** (Unicode) representation of *number*. Octal digits range from 0 to 7.

If you specify *digits* > 0, the result string will be exactly that length. It will be truncated or padded with zeros on the left, if necessary.

The length of the returned string will not be longer than the maximum number of digits required for the type of *number* (3 characters for **Byte**, 6 for **Short**, 11 for **Long**, and 22 for **LongInt**)

Example

```
Print WOct(54321)
Print WOct(54321, 4)
Print WOct(54321, 8)
```

will produce the output:

```
152061
2061
00152061
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__woct`.

Platform Differences

- Unicode strings are not supported in the DOS port of FreeBASIC.

Differences from QB

- In QBASIC Unicode was not supported.

See also

- [WBin](#)
- [WHex](#)

Write



Outputs a comma-separated list of values to the screen

Syntax

```
write [ expressionlist ]
```

Parameters

expressionlist

Comma-separated list of items to print

Description

Outputs the values in *expressionlist* to the screen. The values are separated with commas, and strings are enclosed in double quotes. Numeric values with an absolute value of less than one are prefixed with a zero (0) if none is given (e.g. 0.5, -0.123). Floating-point numbers with absolute values greater than or equal to 10^{16} , or with absolute values greater than 0 and less than 10^{-5} are printed in scientific notation (e.g. $1.8e+019$, $3e-005$)

If no expression list is given, `write` outputs a carriage return.

Example

```
Dim i As Integer = 10
Dim d As Double = 123.456
Dim s As String = "text"

Write 123, "text", -.45600
Write
Write i, d, s
```

will produce the output:

```
123, "text", -0.456
```

```
10, 123.456, "text"
```

Differences from QB

- QBASIC might print format floating-point values in slightly different ways.

See also

- `Write #`
- `(Print | ?)`

Write



Outputs a comma-separated list of values to a text file or device

Syntax

```
Write # filename , [ expressionlist ]
```

Parameters

filename

File number of an open file or device opened for **Output** or **Append**.

expressionlist

Comma-separated list of items to print

Description

Outputs the values in *expressionlist* to the text file or device bound to *filename*. The values are separated with commas, and strings are enclosed in double quotes. Numeric values greater than zero (0) and less than one (1) are output with a zero (0) if none is given (e.g., a value of -.123 will be output as -0.123). Extra zeroes are truncated.

If no expression list is given, `write #` outputs a carriage return (note the comma after *filename* is still necessary, even if no expression list is given). The purpose of `write #` is to create a file that can be read back by using `input #`.

Example

```
Const filename As String = "file.txt"

Dim filenum As Integer = FreeFile()
If 0 <> Open(filename, For Output, As filenum) Then
    Print "error opening " & filename & " for output"
End If
End If

Dim i As Integer = 10
```

```
Dim d As Double = 123.456
Dim s As String = "text"

Write #filenum, 123, "text", -.45600
Write #filenum,
Write #filenum, i, d, s
```

will produce the file:

```
123,"text",-0.456
10,123.456,"text"
```

Differences from QB

- None

See also

- [Write](#)
- [\(Print | ?\) #](#)
- [Input #](#)

Write (File Access)



File access specifier

Syntax

Open *filename* As String For Binary Access **Write** As #*filenum* As Integer

Description

Specifier for the **Access** clause in the **open** statement. **write** specifies that the file is accessible for output.

Example

See example at [Access](#)

Differences from QB

- None known.

See also

- [Access](#)
- [Open](#)

WSpace



Creates a **wstring** of a given length filled with spaces (" ")

Syntax

```
Declare Function WSpace( ByVal count As Integer ) As WString
```

Usage

```
result = WSpace( count )
```

Parameters

count

An integer type specifying the length of the string to be created.

Return Value

The created **wstring**. An empty string will be returned if *count* <= 0.

Description

wspace creates a wstring (wide character string- Unicode) with the specified number of spaces.

Example

```
Dim a As WString * 10
a = "x" + WSpace(3) + "x"
Print a ' prints: x   x
```

Platform Differences

- Unicode strings are not supported in the DOS port of FreeBASIC.

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__wspace`.

Differences from QB

- New to FreeBASIC

See also

- `Space`
- `WString`

Returns a wide-character string representation of a number or ASCII character.

Syntax

```
Declare Function WStr ( ByVal n As Byte ) As WString
Declare Function WStr ( ByVal n As UByte ) As WString
Declare Function WStr ( ByVal n As Short ) As WString
Declare Function WStr ( ByVal n As UShort ) As WString
Declare Function WStr ( ByVal n As Long ) As WString
Declare Function WStr ( ByVal n As ULong ) As WString
Declare Function WStr ( ByVal n As LongInt ) As WString
Declare Function WStr ( ByVal n As ULongInt ) As WString
Declare Function WStr ( ByVal n As Single ) As WString
Declare Function WStr ( ByVal n As Double ) As WString
Declare Function WStr ( ByRef str As Const String ) As WString
Declare Function WStr ( ByVal str As Const WString Ptr ) As WString
```

Usage

```
result = WStr( number )
or
result = WStr( string )
```

Parameters

number

Numeric expression to convert to a wide-character string.

string

String expression to convert to a wide-character string.

Return Value

Returns the wide-character representation of the numeric or string expression.

Description

`wstr` converts numeric variables to their wide-character string representation.

`wstr` also converts ASCII character strings to Unicode character strings. If the input string is returned unmodified.

Example

```
#if defined( __FB_WIN32__ )
#include "windows.bi"
#endif

Dim zs As ZString * 20
Dim ws As WString * 20

zs = "Hello World"
ws = WStr(zs)

#if defined( __FB_WIN32__ )

MessageBox(null, ws, WStr("Unicode 'Hello World'"))

#else

Print ws
Print WStr("Unicode 'Hello World'")

#endif
```

Platform Differences

- DOS does not support `wstr`.

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- `str`
- `wstring`

WString



Standard data type: wide character string

Syntax

```
Dim variable As WString * size  
Dim variable As WString Ptr
```

Description

A `wstring` is a fixed-size array of wide-chars that never overflows if the never resize unless it's a pointer and `Allocate/Reallocate/Deallocate`. FreeBASIC avoids any overflow that could occur on assignment, by tr

The end of the string is marked by the character 0 automatically added character must never be part of a `wstring` or the content will be truncated created, and the length will be calculated by scanning the string for th

In a `wstring`, `Len` returns the size of the contained string and `sizeof` re the size is known by the compiler, i.e. a fixed-size `wstring` variable is j function argument.

This type is provided for support non-Latin based alphabets. Any intrinsic any string operator.

When processing source files, FreeBASIC can parse ASCII files with 16BE, UTF-32LE and UTF-32BE.

The FreeBASIC text file functions can read and write Unicode files in the file is opened. The text is automatically converted to the internal encoding write.

`sizeof(wstring)` returns the number of bytes used by a `wstring` character

Example

```
Dim As WString * 13 str1 => "hello, world"  
Print str1
```

```
Print Len(str1)      'returns 12, the length of the
Print SizeOf(str1)  'returns 13 * sizeof(wstring),
```

```
Dim As WString Ptr str2
str2 = Allocate( 13 * Len(WString) )
*str2 = "hello, world"
Print *str2
Print Len(*str2)    'returns 12, the length of t
```

Platform Differences

Support for wstrings relies in the C runtime library available in the plat

- Unicode is not supported in the DOS port of FreeBASIC will behave as standard ASCII **zstrings**
- On Win32 wstrings are encoded in UCS-2 and a character as FreeBASIC doesn't bother with surrogates introduced understands with a character may not represent a full c
- On Linux wstrings are encoded in UCS-4 and a character

Dialect Differences

- Not available in the **-lang qb** dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- **string** (data type)
- **zstring** (data type)
- **wstring** (data type)
- **string** (function)

- `WString` (function)
- `WSpace`
- `WStr`
- `WChr`
- `WBin`
- `WHex`
- `WOct`
- `Winput()`

Wstring (Function)



Fills a `wstring` with a certain length of a certain wide character

Syntax

```
Declare Function WString ( ByVal count As Integer, ByVal ch_code  
As Long ) As WString  
Declare Function WString ( ByVal count As Integer, ByRef ch As  
Const WString ) As WString
```

Usage

```
result = WString( count, ch_code )  
or  
result = WString( count, ch )
```

Parameters

count

An **Integer** specifying the length of the string to be created.

ch_code

A **Long** specifying the Unicode char to be used to fill the string.

ch

A **wstring** whose first character is to be used to fill the string.

Return Value

The created **wstring**. An empty string will be returned if either *ch* is an empty string, or *count* \leq 0.

Description

wstring generates a temporary **wstring** filled with *count* copies of a **Unicode** character. This string can be printed or assigned to a previously **Dimed** **wstring**.

Example

```
Print WString( 4, 934 )  
Print WString( 5, WStr("Indeed") )
```

End 0

ΦΦΦΦ
IIII

Platform Differences

- Unicode strings are not supported in the DOS port of FreeBASIC.

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__wstring`.

Differences from QB

- QBasic does not support **Unicode**

See also

- **String** (data type)
- **WSpace**
- **WString** (data type)

Operator Xor (Exclusive Disjunction)



Returns the bitwise-xor (exclusive disjunction) of two numeric values

Syntax

```
Declare Operator Xor ( ByRef lhs As T1, ByRef rhs As T2 ) As Ret
```

Usage

```
result = lhs Xor rhs
```

Parameters

lhs

The left-hand side expression.

T1

Any numeric or boolean type.

rhs

The right-hand side expression.

T2

Any numeric or boolean type.

Ret

A numeric or boolean type (varies with *T1* and *T2*).

Return Value

Returns the bitwise-xor of the two operands.

Description

This operator returns the bitwise-exclusion of its operands, a logical or conversion of a boolean to an integer, false or true boolean value bec

The truth table below demonstrates all combinations of a boolean-exc

Lhs Bit	Rhs Bit	Result
0	0	0
1	0	1
0	1	1

1	1	0
---	---	---

No short-circuiting is performed - both expressions are always evaluated.

The return type depends on the types of values passed. **Byte**, **UByte** and **Short** types differ only in signedness, then the return type is the same as the left and right-hand side types are both **Boolean**, the return type is also **Boolean**.

This operator can be overloaded for user-defined types.

Example

```
' Using the XOR operator on two numeric values
Dim As UByte numeric_value1, numeric_value2
numeric_value1 = 15 '00001111
numeric_value2 = 30 '00011110

'Result = 17 = 00010001
Print numeric_value1 Xor numeric_value2
Sleep
```

```
' Using the XOR operator on two conditional expressions
Dim As UByte numeric_value1, numeric_value2
numeric_value1 = 10
numeric_value2 = 15

If numeric_value1 = 10 Xor numeric_value2 = 20 Then
    Print "The result is 20"
Else
    Print "The result is not 20"
End If

Sleep

' This will output "Numeric_Value1 equals 10 or Numeric_Value2 equals 15"
' because only the first condition of the IF state is true.
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- None

See also

- [Operator Truth Tables](#)

Xor



Parameter to the **Put** graphics statement which uses a bit-wise **xor** as the

Syntax

```
Put [ target, ] [ STEP ] ( x,y ), source [ ,( x1,y1 )-( x2,y2 )
```

Parameters

Xor
Required.

Description

The **xor** method combines each source pixel with the corresponding color of the destination using the bit-wise **xor** function. The result of this is output as the destination. This method works in all graphics modes. There is no mask color, although **(RGBA(0, 0, 0, 0)** in full-color modes) will have no effect, because of

In full-color modes, each component (red, green, blue and alpha) is 8 bits, so the operation can be made to only affect some of the channels: all the values of the other channels are set to 0.

Example

```
'open a graphics window
ScreenRes 320, 200, 16

'create a sprite containing a circle
Const As Integer r = 32
Dim c As Any Ptr = ImageCreate(r * 2 + 1, r * 2 + 1, c, (r, r), r, RGBA(255, 255, 255, 0), , , 1

'put the three sprites, overlapping each other in
Put (146 - r, 108 - r), c, Xor
Put (174 - r, 108 - r), c, Xor
Put (160 - r, 84 - r), c, Xor
```

```
''free the memory used by the sprite  
ImageDestroy c
```

```
''pause the program before closing  
Sleep
```



Differences from QB

- None

See also

- Xor
- Put (Graphics)

Gets the year from a **Date Serial**

Syntax

```
Declare Function Year ( ByVal date_serial As Double ) As Long
```

Usage

```
#include "vbcompat.bi"  
result = Year( date_serial )
```

Parameters

date_serial
the date

Return Value

Returns the year from a variable containing a date in **Date Serial** form

Description

The compiler will not recognize this function unless `vbcompat.bi` is inc

Example

```
#include "vbcompat.bi"  
  
Dim a As Double = DateSerial (2005, 11, 28) + Time  
  
Print Format(a, "yyyy/mm/dd hh:mm:ss "); Year(a)
```

Differences from QB

- Did not exist in QB. This function appeared in PDS and VBDO:

See also

- **Date Serials**

ZString



Standard data type: 8 bit character string

Syntax

```
Dim variable As ZString * size  
Dim variable As ZString Ptr
```

Description

A **zstring** is a C-style fixed-size array of chars. It has no descriptor so pass it as an argument to functions. When the variable has a fixed *size* overflow that could occur on assignment, by truncating the contents to

A **zstring Ptr** can point to a standard **zstring**, also can be used to im **zstring**, in this case **Allocate/Reallocate/Deallocate** must be used to the user to avoid overflows .

The end of the string is marked by a null character (  ASCII). This is a FreeBASIC string handling functions. A null character will be appended the length will be calculated by scanning the string for the first null character (**Chr( )**) may never be contained in the text of a **zstring** or the rest of t

In a **zstring**, **Len** returns the size of the contained string and **SizeOf** re **zstring**. **SizeOf** only works if the size is known by the compiler, i.e. a f passed directly, not as a dereferenced pointer or a **ByRef** function argu

This type is provided for easy interfacing with C libraries and to also re that can't be managed through pointers. Any intrinsic string functions l too, plus any string operator.

Example

```
Dim As ZString * 13 str1 => "hello, world"  
Print str1  
Print Len(str1)      'returns 12, the size of the s  
Print SizeOf(str1)  'returns 13, the size of the v
```

```
Dim As ZString Ptr str2
str2 = Allocate( 13 )
*str2 = "hello, world"
Print *str2
Print Len(*str2)      'returns 12, the size of the
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the

Differences from QB

- New to FreeBASIC

See also

- [String](#)
- [WString](#)

Functional Keyword List



List of FreeBASIC keywords sorted by the function they perform.

Arrays

- Erase
- LBound
- ReDim
- Preserve
- UBound

Bit manipulation

- Bit
- BitReset
- BitSet
- HiByte
- HiWord
- LoByte
- LoWord

Compiler switches

- DefByte
- DefDbl
- DefInt
- DefLng
- Deflongint
- DefShort
- DefSng
- DefStr
- DefUByte
- DefUInt
- Defulongint

Miscellaneous

- Asm
- Data
- End (Block)
- Is (Run-Time Type Information Operator)
- Let
- OffsetOf
- Option()
- To
- Read
- Rem
- Restore
- SizeOf
- Swap
- TypeOf

Modularizing

- Common
- DyLibFree
- DyLibLoad
- DyLibSymbol
- Export
- Extern
- Extern...End Extern
- Import
- Namespace
- Private

- DefUShort
- Option Base
- Option ByVal
- Option Dynamic
- Option Escape
- Option Explicit
- Option Gosub
- Option Nogosub
- Option NoKeyword
- Option Private
- Option Static

Console

- Beep
- Cls
- Color
- CsrLin
- Locate
- Open Cons
- Open Err
- Open Pipe
- Open Scrn
- Pos
- Print
- ?
- Print Using
- ? Using
- Screen (Console)
- Spc
- Tab
- View (Console)
- Width

- Public
- Using (Namespaces)

Multithreading

- CondBroadcast
- CondCreate
- CondDestroy
- CondSignal
- CondWait
- MutexCreate
- MutexDestroy
- MutexLock
- MutexUnlock
- Threadcall
- ThreadCreate
- Threaddetach
- ThreadWait

OS / shell

- Chain
- ChDir
- Command
- CurDir
- Dir
- End (Statement)
- Environ
- Exec
- ExePath
- FileAttr
- FileCopy
- FileDateTime
- FileExists

- Write

Data types and declarations

- Boolean
- Byte
- As
- Dim
- Const
- Const (Qualifier)
- Double
- Enum
- Extends
- Integer
- Long
- LongInt
- Object
- Scope
- Shared
- Short
- String
- Single
- Static
- Type
- Type (Alias)
- Type (Temporary)
- UByte
- UInteger
- Ulong
- ULongInt
- Union
- Unsigned
- UShort

- FileLen
- Isredirected
- Kill
- Mkdir
- Name
- Rmdir
- Run
- SetEnviron
- Shell
- System
- WindowTitle

Pointers

- Pointer
- ProcPtr
- Ptr
- SAdd
- StrPtr
- VarPtr

Predefined symbols

- __DATE__
- __Date_Iso__
- __Fb_64Bit__
- __FB_ARGC__
- __FB_ARGV__
- __Fb_Arm__
- __FB_BIGENDIAN__
- __FB_BUILD_DATE__
- __FB_CYGWIN__
- __FB_DARWIN__
- __FB_DEBUG__

- Var
- With
- WString
- ZString

Date and time

- Date
- DateAdd
- DateDiff
- DatePart
- DateSerial
- DateValue
- Day
- Hour
- IsDate
- Minute
- Month
- MonthName
- Now
- Second
- SetDate
- SetTime
- Time
- TimeSerial
- TimeValue
- Timer
- Year
- Weekday
- WeekdayName

Debug support

- Assert

- __FB_DOS__
- __FB_ERR__
- __FB_FREEBSD__
- __FB_LANG__
- __FB_LINUX__
- __FB_MAIN__
- __FB_MIN_VERSION__
- __FB_NETBSD__
- __FB_OPENBSD__
- __FB_OPTION_BYVAL__
- __FB_OPTION_DYNAMIC__
- __FB_OPTION_ESCAPE__
- __FB_OPTION_EXPLICIT__
- __Fb_Option_Gosub__
- __FB_OPTION_PRIVATE__
- __FB_OUT_DLL__
- __FB_OUT_EXE__
- __FB_OUT_LIB__
- __FB_OUT_OBJ__
- __FB_SIGNATURE__
- __FB_SSE__
- __FB_VERSION__
- __FB_VER_MAJOR__
- __FB_VER_MINOR__
- __FB_VER_PATCH__
- __FB_WIN32__
- __FB_XBOX__
- __FILE__
- __FILE_NQ__
- __FUNCTION__
- __FUNCTION_NQ__

- AssertWarn
- Stop

Error handling

- Erfn
- Erl
- Ermn
- Err
- Error
- Local
- On Error
- Resume
- Resume Next

Files

- Access
- Append
- Binary
- BLoad
- BSave
- Close
- Encoding
- EOF
- FreeFile
- Get # (File I/O)
- Input (File I/O)
- Input #
- Line Input #
- LOC
- Lock
- LOF
- Open

- __FB_MT__
- __LINE__
- __PATH__
- __TIME__
- False
- True

Preprocessor

- #Assert
- #define
- #else
- #elseif
- #endif
- #endmacro
- #error
- #if
- #ifdef
- #ifndef
- #inclid
- #include
- #libpath
- #lang
- #line
- #macro
- #pragma
- #print
- #undef
- defined
- Once

Procedures

- ...

- Output
- Print #
- ? #
- Put # (File I/O)
- Random
- Read (File Access)
- Read Write (File Access)
- Reset
- Seek (Statement)
- Seek (Function)
- Unlock
- Write #
- Write (File Access)
- Abstract (Member)
- Alias
- Any
- Base (Initialization)
- Base (Member Access)
- Byref (Parameters)
- Byref (Function Results)
- ByVal
- Call
- cdecl
- Const (Member)
- Constructor
- Constructor (Module)
- Destructor
- Destructor (Module)

Graphics

- Add (Graphics Put)
- Alpha (Graphics Put)
- And (Graphics Put)
- Circle
- Cls
- Color
- Custom (Graphics Put)
- Draw
- Draw String
- Event (Message Data From ScreenEvent)
- Flip
- Get (Graphics)
- ImageConvertRow
- ImageCreate
- Declare
- Function
- Function (Member)
- Lib
- Naked
- Operator
- Overload
- Override
- pascal
- Private (Member)
- Protected (Member)
- Property
- Public (Member)
- Static (Member)
- Sub
- Sub (Member)

- ImageDestroy
 - ImageInfo
 - Line
 - Or (Graphics Put)
 - Paint
 - Palette
 - PCopy
 - PMap
 - Point
 - Pointcoord
 - PReset
 - PSet
 - Pset (Graphics Put)
 - Put (Graphics)
 - RGB
 - RGBA
 - Screen
 - ScreenControl
 - ScreenCopy
 - ScreenEvent
 - ScreenInfo
 - ScreenGLProc
 - ScreenList
 - ScreenLock
 - ScreenPtr
 - ScreenRes
 - ScreenSet
 - ScreenSync
 - ScreenUnlock
 - Trans (Graphics Put)
 - View (Graphics)
 - stdcall
 - This
 - va_arg
 - va_first
 - va_next
 - Virtual (Member)
- Program flow**
- Continue
 - Case
 - Do
 - Do...Loop
 - Else
 - Elself
 - End If
 - Exit
 - GoSub
 - Goto
 - If...Then
 - If
 - Is (Select Case)
 - For
 - For...Next
 - Loop
 - Next
 - On...Gosub
 - On...Goto
 - Return
 - Select Case
 - Sleep
 - Step
 - Then

- Window
- Xor (Graphics Put)

Hardware access

- Inp
- Out
- Wait
- Open Com
- Open Lpt
- Lpt
- Lpos
- LPrint

Assignment Operators

- [=]>] (Assignment)
- &= (Concatenate And Assign)
- += (Add And Assign)
- -= (Subtract And Assign)
- *= (Multiply And Assign)
- /= (Divide And Assign)
- \= (Integer Divide And Assign)
- ^= (Exponentiate And Assign)
- Mod= (Modulus And Assign)
- And= (Conjunction And Assign)
- Eqv= (Equivalence And Assign)

- Until
- Wend
- While
- While...Wend

String functions

- InStr
- InStrRev
- LCase
- Left
- Len
- LSet
- LTrim
- Mid (Statement)
- Mid (Function)
- Right
- RSet
- RTrim
- Space
- String (Function)
- Trim
- UCase
- WSpace
- Wstring (Function)

String and number conversion

- Asc
- Bin
- Chr
- CVD
- CVI
- CVL

- Imp= (Implication And Assign)
- Or= (Inclusive Disjunction And Assign)
- Xor= (Exclusive Disjunction And Assign)
- Shl= (Shift Left And Assign)
- Shr= (Shift Right And Assign)
- Let (Assignment)
- Let() (Assignment)
- CVLongInt
- CVS
- CVShort
- Format
- Hex
- MKD
- MKI
- MKL
- MKLongInt
- MKS
- MKShort
- Oct
- Str
- Val
- ValLng
- ValInt
- ValUInt
- ValULng
- WBin
- WChr
- WHex
- WOct
- WStr

Arithmetic Operators

- + (Add)
- - (Subtract)
- * (Multiply)
- / (Divide)
- \ (Integer Divide)
- ^ (Exponentiate)
- Mod (Modulus)
- - (Negate)
- Shl (Shift Left)
- Shr (Shift Right)

Bitwise operators

- And
- Eqv
- Imp
- Or
- Not

Type casting/conversion

- Cast
- Cbool
- CByte
- CDbI
- CInt
- CLng
- CLngInt

- Xor

Short Circuit operators

- AndAlso
- OrElse

Math

- Abs
- Acos
- Asin
- Atan2
- Atn
- Cos
- Exp
- Fix
- Frac
- Int
- Log
- Randomize
- Rnd
- Sgn
- Sin
- Sqr
- Tan

Memory

- Allocate
- CAllocate
- Clear
- Deallocate
- Field
- Fre
- Peek

- CPtr
- CShort
- CSign
- CSng
- CUByte
- CUInt
- CULng
- CULngInt
- CUnsg
- CUShort

User input

- GetJoystick
- GetKey
- GetMouse
- Inkey
- Input
- Input (Statement)
- Line Input
- MultiKey
- SetMouse
- Stick
- Strig
- WInput

- **Poke**
- **Reallocate**

Meta Commands

- **\$Dynamic**
- **\$Static**
- **\$Include**
- **\$Lang**

Operator [=]>] (Assign)



Assigns a value to a variable

Syntax

Declare Operator Let (**ByRef** *lhs* **As** *T1*, **ByRef** *rhs* **As** *T2*)

Usage

lhs = *rhs*

or

lhs => *rhs* (from fbc version 0.90)

or, in the QB dialect,

[**Let**] *lhs* = *rhs*

or

[**Let**] *lhs* => *rhs* (from fbc version 0.90)

Parameters

lhs

The variable to assign to.

T1

Any numeric, boolean, string or pointer type.

rhs

The value to assign to *lhs*.

T2

Any type convertible to *T2*.

Description

This operator assigns the value of its right-hand side operand (*rhs*) to its left-hand side operand (*lhs*). The right-hand side operand must be implicitly convertible to the left-hand side type (*T1*) (for conversion of a boolean to an integer, false or true boolean value becomes 0 or -1 integer value). For example, you cannot assign a numeric value to a string type; to do that, first convert the numeric value to a string using **str** or **wstr**.

Assignment between arrays is not supported presently.

Avoid confusion with **Operator = (Equal)**, which also uses the '=' symbol.

For this purpose and for solving some cases of ambiguity of the parser (see **Byref (Function Results)**), the alternative symbol '=>' can be used for assignments (in place of '=') from fbc version 0.90 (same as already for the initializers).

Note: the '=>' symbol has been chosen against '<=' (already the operator 'Less Than Or Equal') and ':=' (':' used as statement separator).

This operator can be overloaded for user-defined types.

Example

```
Dim i As Integer
i = 420      ' <- this is the assignment operator

If i = 69 Then ' <-
this is the equivalence operator
  Print "ERROR: i should equal 420"
  End -1
End If

Print "All is good."
End 0
```

```
' compile with -lang fblite or qb

#lang "fblite"

Dim i As Integer
Let i = 300 ' <-alternate syntax
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.
- In the *-lang qb* dialect, an assignment expression can be preceded by the `Let` keyword.

Differences from QB

- None

See also

- `Operator = (Equal)`
- `Operator Let (Assignment)`
- `Swap`

Operator &= (Concatenate And Assign)



Appends and assigns a string onto another string

Syntax

```
Declare Operator &= ( ByRef lhs As String, ByRef rhs As T2 )
Declare Operator &= ( ByRef lhs As WString, ByRef rhs As T2 )
```

Usage

```
lhs &= rhs
```

Parameters

lhs

The string to assign to.

rhs

The value to append to *lhs*.

T2

Any numeric, string or user-defined type that can be converted to a string.

Description

This operator appends one string onto another. The right-hand side expression (*rhs*) is converted to a string before concatenation. It is functionally equivalent to,

```
lhs = lhs & rhs
```

where the result is assigned back to the left-hand side string.

This operator can be overloaded for user-defined types.

Note: This operator exists in C/C++ with a different meaning - there it performs a bitwise **And=**.

Example

```
Dim s As String = "Hello, "  
s &= " world!"  
Print s
```

will produce the output:

```
Hello, world!
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- New to FreeBASIC

See also

- [Operator & \(String Concatenation With Conversion\)](#)
- [Operator += \(Add And Assign\)](#)

Operator += (Add And Assign)



Adds and assigns a value to a variable

Syntax

```
Declare Operator += ( ByRef lhs As T1, ByRef rhs As T2 )
```

```
Declare Operator += ( ByRef lhs As T Ptr, ByRef rhs As Integer )
```

```
Declare Operator += ( ByRef lhs As String, ByRef rhs As String )  
Declare Operator += ( ByRef lhs As WString, ByRef rhs As WString )
```

Usage

```
lhs += rhs
```

Parameters

lhs

The variable to assign to.

T1

Any numeric type.

rhs

The value to add to *lhs*.

T2

Any numeric type.

T

Any data type.

Description

This operator adds and assigns a value to a variable. It is functionally equivalent to:

```
lhs = lhs + rhs
```

For numeric types, the right-hand side expression (*rhs*) will be converted to the left-hand side type (*T1*).

For string types, this operator is functionally equivalent to **operator &=**

(Concatenate And Assign).

This operator can be overloaded for user-defined types.

Example

```
Dim n As Double
n = 6
n += 1
Print n
Sleep
```

Output:

```
7
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- New to FreeBASIC

See also

- **Operator + (Add)**
- **Mathematical Functions**

Operator -= (Subtract And Assign)



Subtracts and assigns a value to a variable

Syntax

```
Declare Operator -= ( ByRef lhs As T1, ByRef rhs As T2 )
```

```
Declare Operator -= ( ByRef lhs As T Ptr, ByRef rhs As Integer )
```

Usage

```
lhs -= rhs
```

Parameters

lhs

The variable to assign to.

T1

Any numeric type.

rhs

The value to subtract from *lhs*.

T2

Any numeric type.

T

Any data type.

Description

This operator subtracts and assigns a value to a variable. It is functionally equivalent to:

```
lhs = lhs - rhs
```

For numeric types, the right-hand side expression (*rhs*) will be converted to the left-hand side type (*T1*).

This operator can be overloaded for user-defined types.

Example

```
Dim n As Double
n = 6
n -= 2.2
Print n
Sleep
```

Output:

```
3.8
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- New to FreeBASIC

See also

- **Operator - (Subtract)**
- **Mathematical Functions**

Operator *= (Multiply And Assign)



Multiplies and assigns a value to a variable

Syntax

```
Declare Operator *= ( ByRef lhs As T1, ByRef rhs As T2 )
```

Usage

```
lhs *= rhs
```

Parameters

lhs

The variable to assign to.

T1

Any numeric type.

rhs

The value to multiply *lhs* by.

T2

Any numeric type.

Description

This operator multiplies and assigns a value to a variable. It is functionally equivalent to:

```
lhs = lhs * rhs
```

The right-hand side expression (*rhs*) will be converted to the left-hand side type (*T1*).

This operator can be overloaded for user-defined types.

Example

```
Dim n As Double  
n = 6  
n *= 2
```

```
Print n  
Sleep
```

Output:

```
12
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- New to FreeBASIC

See also

- [Operator * \(Multiply\)](#)
- [Mathematical Functions](#)

Operator /= (Divide And Assign)



Divides and assigns a value to a variable

Syntax

```
Declare Operator /= ( ByRef lhs As T1, ByRef rhs As T2 )
```

Usage

```
lhs /= rhs
```

Parameters

lhs

The variable to assign to.

T1

Any numeric type.

rhs

The value to divide *lhs* by.

T2

Any numeric type.

Description

This operator divides and assigns a value to a variable. It is functionally equivalent to:

```
lhs = lhs / rhs
```

This operator can be overloaded for user-defined types.

Example

```
Dim n As Double  
n = 6  
n /= 2.2  
Print n  
Sleep
```

Output:

```
2.727272727272727
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- New to FreeBASIC

See also

- [Operator / \(Divide\)](#)
- [Mathematical Functions](#)

Operator \= (Integer Divide And Assign)



Integer divides and assigns a value to a variable

Syntax

```
Declare Operator \= ( ByRef lhs As T1, ByRef rhs As T2 )
```

Usage

```
lhs \= rhs
```

Parameters

lhs

The variable to assign to.

T1

Any numeric type.

rhs

The value to divide *lhs* by.

T2

Any numeric type.

Description

This operator multiplies and assigns a value to a variable. It is functionally equivalent to:

```
lhs = lhs \ rhs
```

This operator can be overloaded for user-defined types.

Example

```
Dim n As Double  
n = 6  
n \= 2.2  
Print n  
Sleep
```

Output:

3

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- New to FreeBASIC

See also

- **Operator \ (Integer Divide)**
- **Mathematical Functions**

Operator ^= (Exponentiate And Assign)



Exponentiates and assigns a value to a variable

Syntax

```
Declare Operator ^= ( ByRef lhs As Double, ByRef rhs As Double )
```

Usage

```
lhs ^= rhs
```

Parameters

lhs

The variable to assign to.

rhs

The value to exponentiate *lhs* by.

Description

This operator exponentiates and assigns a value to a variable. It is functionally equivalent to:

```
lhs = lhs ^ rhs
```

This operator can be overloaded for user-defined types.

Note: This operator exists in C/C++ with a different meaning - there it performs a Bitwise **xor=**.

Example

```
Dim n As Double
n = 6
n ^= 2
Print n
Sleep
```

Output:

36

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- New to FreeBASIC

See also

- **Operator ^ (Exponentiate)**
- **Mathematical Functions**

Operator Mod= (Modulus And Assign)



Divides a value and assigns the remainder to a variable

Syntax

```
Declare Operator Mod= ( ByRef lhs As Integer, ByRef rhs As Integer )
```

Usage

```
lhs Mod= rhs
```

Parameters

lhs

The variable to assign to.

rhs

The value to divide *lhs* by.

Description

This operator divides two values of **Integer** type and assigns the remainder to its left-hand side (*lhs*) variable. It is functionally equivalent to:

```
lhs = lhs Mod rhs
```

This operator can be overloaded for user-defined types.

Example

```
Dim n As Integer
n = 11
n Mod= 3
' The result is 2
Print n
Sleep
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- New to FreeBASIC

See also

- [Operator + \(Modulus\)](#)
- [Mathematical Functions](#)

Operator And= (Conjunction And Assign)



Performs a bitwise-and (conjunction) and assigns the result to a variable

Syntax

```
Declare Operator And= ( ByRef lhs As T1, ByRef rhs As T2 )
```

Usage

```
lhs And= rhs
```

Parameters

lhs

The variable to assign to.

T1

Any numeric or boolean type.

rhs

The value to perform a bitwise-and (conjunction) with *lhs*.

T2

Any numeric or boolean type.

Description

This operator performs a bitwise-and and assigns the result to a variable (for conversion of a boolean to an integer, false or true boolean value becomes 0 or -1 integer value). It is functionally equivalent to:

```
lhs = lhs And rhs
```

And= compares each bit of its operands, *lhs* and *rhs*, and if both bits are 1, then the corresponding bit in the first operand, *lhs*, is set to 1, otherwise it is set to 0.

And= cannot be used in conditional expressions.

This operator can be overloaded for user-defined types.

Example

```
' Using the AND= operator on two numeric values
Dim As UByte numeric_value1, numeric_value2
numeric_value1 = 15 ' 00001111
numeric_value2 = 30 ' 00011110

numeric_value1 And= numeric_value2

' Result = 14 = 00001110
Print numeric_value1
Sleep
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- New to FreeBASIC

See also

- [And](#)

Operator Eqv= (Equivalence And Assign)



Performs a bitwise-eqv (equivalence) and assigns the result to a variable.

Syntax

```
Declare Operator Eqv= ( ByRef lhs As T1, ByRef rhs As T2 )
```

Usage

```
lhs Eqv= rhs
```

Parameters

lhs

The variable to assign to.

T1

Any numeric or boolean type.

rhs

The value to perform a bitwise-eqv (equivalence) with *lhs*.

T2

Any numeric or boolean type.

Description

This operator performs a bitwise-eqv and assigns the result to a variable (for conversion of a boolean to an integer, false or true boolean value becomes 0 or -1 integer value). It is functionally equivalent to:

```
lhs = lhs Eqv rhs
```

Eqv= compares each bit of its operands, *lhs* and *rhs*, and if both bits are the same (either both 0 or both 1), then the corresponding bit in the first operand, *lhs*, is set to 1, otherwise it is set to 0.

This operator can be overloaded for user-defined types.

Example

```
Dim As UByte a = &b00110011
Dim As UByte b = &b01010101
a Eqv= b
'' Result      a = &b10011001
Print Bin(a)
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- New to FreeBASIC

See also

- [Eqv](#)

Operator Imp= (Implication And Assign)



Performs a bitwise-imp (implication) and assigns the result to a variable

Syntax

```
Declare Operator Imp= ( ByRef lhs As T1, ByRef rhs As T2 )
```

Usage

```
lhs Imp= rhs
```

Parameters

lhs

The variable to assign to.

T1

Any numeric or boolean type.

rhs

The value to perform a bitwise-imp (implication) with *lhs*.

T2

Any numeric or boolean type.

Description

This operator performs a bitwise-imp and assigns the result to a variable (for conversion of a boolean to an integer, false or true boolean value becomes 0 or -1 integer value). It is functionally equivalent to:

```
lhs = lhs Imp rhs
```

Imp is a bitwise operator which is the same as **(Not lhs) Or rhs**. **Imp=** compares each bit of its operands, *lhs* and *rhs*, and if the bit in *lhs* is 0 or the bit in *rhs* is 1, then the corresponding bit in the first operand, *lhs* is set to 1, otherwise it is set to 0.

This operator can be overloaded for user-defined types.

Example

```
Dim As UByte a = &b00110011
Dim As UByte b = &b01010101
a Imp= b
'' Result      a = &b11011101
Print Bin(a)
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- New to FreeBASIC

See also

- [Imp](#)
- [Assignment Operators](#)

Operator Or= (Inclusive Disjunction And Assign)



Performs a bitwise-or (inclusive disjunction) and assigns the result to a variable

Syntax

```
Declare Operator Or= ( ByRef lhs As T1, ByRef rhs As T2 )
```

Usage

```
lhs Or= rhs
```

Parameters

lhs

The variable to assign to.

T1

Any numeric or boolean type.

rhs

The value to perform a bitwise-or (inclusive disjunction) with *lhs*.

T2

Any numeric or boolean type.

Description

This operator performs a bitwise-or and assigns the result to a variable (for conversion of a boolean to an integer, false or true boolean value becomes 0 or -1 integer value). It is functionally equivalent to:

```
lhs = lhs Or rhs
```

or= compares each bit of its operands, *lhs* and *rhs*, and if either bits are 1, then the corresponding bit in the first operand, *lhs*, is set to 1, otherwise it is set to 0.

This operator can be overloaded for user-defined types.

Example

```
Dim As UByte a = &b00110011
Dim As UByte b = &b01010101
a Or= b
'' Result      a = &b01110111
Print Bin(a)
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- New to FreeBASIC

See also

- `Or`

Operator Xor= (Exclusive Disjunction And Assign)



Performs a bitwise-xor (exclusive disjunction) and assigns the result to a variable

Syntax

```
Declare Operator Xor= ( ByRef lhs As T1, ByRef rhs As T2 )
```

Usage

```
lhs Xor= rhs
```

Parameters

lhs

The variable to assign to.

T1

Any numeric or boolean type.

rhs

The value to perform a bitwise-xor (exclusive or) with *lhs*.

T2

Any numeric or boolean type.

Description

This operator performs a bitwise-or and assigns the result to a variable (for conversion of a boolean to an integer, false or true boolean value becomes 0 or -1 integer value). It is functionally equivalent to:

```
lhs = lhs Xor rhs
```

xor= compares each bit of its operands, *lhs* and *rhs*, and if both bits are the same (both 1 or both 0), then the corresponding bit in the first operand, *lhs*, is set to 0, otherwise it is set to 1.

This operator can be overloaded for user-defined types.

Example

```
Dim As UByte a = &b00110011
Dim As UByte b = &b01010101
a Xor= b
'' Result      a = &b01100110
Print Bin(a)
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- New to FreeBASIC

See also

- [Xor](#)

Operator Shl= (Shift Left And Assign)



Shifts left and assigns a value to a variable

Syntax

```
Declare Operator Shl= ( ByRef lhs As Integer, ByRef rhs As Integer )
Declare Operator Shl= ( ByRef lhs As UInteger, ByRef rhs As UInteger )
Declare Operator Shl= ( ByRef lhs As LongInt, ByRef rhs As LongInt )
Declare Operator Shl= ( ByRef lhs As ULongInt, ByRef rhs As ULongInt )
```

Usage

```
lhs shl= rhs
```

Parameters

lhs

The variable to assign to.

rhs

The value to shift *lhs* left by.

Description

This operator shifts the bits in its left-hand side (*lhs*) parameter a number of times specified by its right-hand side (*rhs*) parameter, and assigns the result to *lhs*. It is functionally equivalent to:

```
lhs = lhs Shl rhs
```

This operator can be overloaded for user-defined types.

Example

```
Dim i As Integer
i = &b00000011    '' = 3
i Shl= 3          '' = i*2^3
```

```
' ' Result: 11000          24          24
Print Bin(i), i, 3*2^3
Sleep
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__sh1=`.

Differences from QB

- New to FreeBASIC

See also

- **Operator Shl (Shift Left)**
- **Operator Shr= (Shift Right And Assign)**
- **Mathematical Functions**

Operator Shr= (Shift Right And Assign)



Shifts right and assigns a value to a variable

Syntax

```
Declare Operator Shr= ( ByRef lhs As Integer, ByRef rhs As Integer )
Declare Operator Shr= ( ByRef lhs As UInteger, ByRef rhs As UInteger )
Declare Operator Shr= ( ByRef lhs As LongInt, ByRef rhs As LongInt )
Declare Operator Shr= ( ByRef lhs As ULongInt, ByRef rhs As ULongInt )
```

Usage

lhs shr= *rhs*

Parameters

lhs

The variable to assign to.

rhs

The value to shift *lhs* right by.

Description

This operator shifts the bits in its left-hand side (*lhs*) parameter a number of times specified by its right-hand side (*rhs*) parameter, and assigns the result to *lhs*. It is functionally equivalent to:

lhs = *lhs* Shr *rhs*

This operator can be overloaded for user-defined types.

Example

```
Dim i As Integer
i = &b00011000    '' = 24
i Shr= 3          '' = i\2^3
```

```
' ' Result: 11          3          3
Print Bin(i), i, 24\2^3
Sleep
```

Dialect Differences

- Not available in the *-lang qb* dialect unless referenced with the alias `__shr=`.

Differences from QB

- New to FreeBASIC

See also

- **Operator Shr (Shift Right)**
- **Operator Shl= (Shift Left And Assign)**
- **Mathematical Functions**

Operator Let (Assign)



Indicates the assignment operator when overloading **Operator = (Assign)**

Syntax

```
{ Type | Class | Union | Enum } typename  
Declare Operator Let ( [ ByRef | ByVal ] rhs As datatype )  
End { Type | Class | Union }
```

```
Operator typename.Let ( [ ByRef | ByVal ] rhs As datatype )
```

Usage

```
lhs = rhs
```

or

```
lhs => rhs (from fbc version 0.90)
```

Parameters

typename

name of the **Type**, **Class**, **Union**, Or **Enum**

lhs

The variable to assign to.

rhs

The value to assign.

Description

Let is used to overload the **Operator = [>] (Assignment)** operator and to

lhs = [>] *rhs* will assign the *rhs* to *lhs* by invoking the **Let** operator procedure. This includes the case of an object returned from a function by value, by assignment.

Assigning one array is not supported presently.

An operator **Let** (assign) must be defined if the shallow implicit copy is for dynamically allocated memory or other resources which need to be specially handled (e.g. dynamically allocated memory, the implicit assignment operator will simply perform the copy of data).

Note: It is safe to do a check for self-assignment at the top of the **Let** procedure.

address of 'rhs' parameter) to avoid object destruction if previously alloc

Example

```
Type UDT
  Public:
    Declare Constructor (ByVal zp As Const ZString
    Declare Operator Let (ByRef rhs As UDT)
    Declare Function getString () As String
    Declare Destructor ()
  Private:
    Dim zp As ZString Ptr
End Type

Constructor UDT (ByVal zp As Const ZString Ptr)
  This.zp = CAllocate(Len(*zp) + 1)
  *This.zp = *zp
End Constructor

Operator UDT.Let (ByRef rhs As UDT)
  If @This <> @rhs Then ' check for self-assignme
    Deallocate(This.zp)
    This.zp = CAllocate(Len(*rhs.zp) + 1)
    *This.zp = *rhs.zp
  End If
End Operator

Function UDT.getString () As String
  Return *This.zp
End Function

Destructor UDT ()
  Deallocate(This.zp)
End Destructor

Dim u As UDT = UDT("")
u = Type<UDT>("Thanks to the overloading operator L
```

```
Print u.getString  
Sleep
```

Output:

```
Thanks to the overloading operator Let (assign)
```

Dialect Differences

- In the *-lang qb* and *-lang fblite* dialects, this operator cannot be
- In the *-lang qb* and *-lang fblite* dialects, an assignment express

Differences from QB

- None

See also

- `Let`
- `Operator Let() (Assignment)`
- `Operator =[>] (Assignment)`
- `Operator = (Equal)`

Operator Let() (Assignment)



Assigns fields of a user defined type to a list of variables

Syntax

```
Let( variable1 [, variable2 [, ... ]] ) = UDT_var  
or
```

```
Let( variable1 [, variable2 [, ... ]] ) => UDT_var (from fbc version 0.90)
```

Parameters

```
variable1 [, variable2 [, ... ]]
```

Comma separated list of variables to receive the values of the *UDT* variable's fields.

```
UDT_var
```

A user defined type variable.

Description

Assigns the values from the *UDT_var* variable's fields to the list of variables.

Union is not supported.

Example

```
Type Vector3D  
  x As Double  
  y As Double  
  z As Double  
End Type  
  
Dim a As Vector3D = ( 5, 7, 9 )  
  
Dim x As Double, y As Double  
  
' Get the first two fields only  
Let( x, y ) = a
```

```
Print "x = "; x  
Print "y = "; y
```

Output:

```
x = 5  
y = 7
```

Dialect Differences

- Only available in the *-lang fb* dialect.

Differences from QB

- New to FreeBASIC

See also

- [Let](#)
- [Operator \[=\]> \(Assignment\)](#)
- [Operator Let \(Assignment\)](#)

Operator + (Addition)



Sums two expressions

Syntax

```
Declare Operator + ( ByRef lhs As Integer, ByRef rhs As Integer  
As Integer
```

```
Declare Operator + ( ByRef lhs As UInteger, ByRef rhs As UInteger  
) As UInteger
```

```
Declare Operator + ( ByRef lhs As LongInt, ByRef rhs As LongInt  
As LongInt
```

```
Declare Operator + ( ByRef lhs As ULongInt, ByRef rhs As ULongInt  
) As ULongInt
```

```
Declare Operator + ( ByRef lhs As Single, ByRef rhs As Single )  
As Single
```

```
Declare Operator + ( ByRef lhs As Double, ByRef rhs As Double )  
As Double
```

```
Declare Operator + ( ByRef lhs As T Pointer, ByRef rhs As Integer  
) As T Pointer
```

```
Declare Operator + ( ByRef rhs As Integer, ByRef lhs As T Pointer  
) As T Pointer
```

```
Declare Operator + ( ByRef lhs As T, ByRef rhs As Integer ) As T
```

```
Declare Operator + ( ByRef lhs As Integer, ByRef rhs As T ) As T
```

Usage

```
result = lhs + rhs
```

Parameters

lhs

The left-hand side expression to sum.

rhs

The right-hand side expression to sum.

T

Any pointer type.

Return Value

Returns the sum of two expressions.

Description

When the left and right-hand side expressions are numeric values, `operator + (Add)` returns the sum of the two values.

When the left and right-hand side expressions are string values, `operator + (Add)` concatenates the two strings and returns the result.

If an integral value n is added to a T `Pointer` type, the operator performs pointer arithmetic on the address, returning the memory position of a T value, n indices away (assuming n is within bounds of a contiguous array of T values). This behaves differently from numeric addition, because the `Integer` value is scaled by `SizeOf(T)`.

Neither operand is modified in any way.

This operator can be overloaded to accept user-defined types.

Example

```
Dim n As Single
n = 4.75 + 5.25
Print n
```

will produce the output:

```
10
```

Dialect Differences

- In the `-lang qb` dialect, this operator cannot be overloaded.

Differences from QB

- None

See also

- **Operator + (String Concatenation)**
- **Mathematical Functions**

Operator - (Subtract)



Subtracts two expressions

Syntax

Declare Operator - (ByRef lhs As Integer, ByRef rhs As Integer
As Integer

Declare Operator - (ByRef lhs As UInteger, ByRef rhs As UInteger
) As UInteger

Declare Operator - (ByRef lhs As LongInt, ByRef rhs As LongInt
As LongInt

Declare Operator - (ByRef lhs As ULongInt, ByRef rhs As ULongInt
) As ULongInt

Declare Operator - (ByRef lhs As Single, ByRef rhs As Single)
As Single

Declare Operator - (ByRef lhs As Double, ByRef rhs As Double)
As Double

Declare Operator - (ByRef lhs As T Pointer, ByRef rhs As T
Pointer) As Integer

Declare Operator - (ByRef lhs As T Pointer, ByRef rhs As Integer
) As T Pointer

Declare Operator - (ByRef lhs As T, ByRef rhs As T) As Integer

Declare Operator - (ByRef lhs As T, ByRef rhs As Integer) As T

Declare Operator - (ByRef lhs As Integer, ByRef rhs As T) As T

Usage

result = *lhs* - *rhs*

Parameters

lhs

The left-hand side expression to subtract from.

rhs

The right-hand side expression to subtract.

T

Any pointer type.

Return Value

Returns the subtraction of two expressions.

Description

When the left and right-hand side expressions are numeric values, `operator -` (**Subtract**) returns the subtraction of the two values.

If the left and right-hand side expressions are both of the τ **Pointer** type, for some type τ , the operator performs pointer subtraction on the address, returning the result. This is different from numeric subtraction because the difference is divided by `SizeOf(τ)`.

If an integral value n is subtracted from a τ **Pointer** type, the operator performs pointer arithmetic on the address, returning the memory position of a τ value, n indices before (assuming $(-n)$ is within bounds of a contiguous array of τ values). This behaves differently from numeric subtraction, because the **Integer** value is scaled by `SizeOf(τ)`.

Neither operand is modified in any way.

This operator can be overloaded to accept user-defined types.

Example

```
Dim n As Single
n = 4 - 5
Print n
```

will produce the output:

```
-1
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- None

See also

- **Mathematical Functions**

Operator * (Multiply)



Multiplies two numeric expressions

Syntax

```
Declare Operator * ( ByRef lhs As Integer, ByRef rhs As Integer
As Integer
Declare Operator * ( ByRef lhs As UInteger, ByRef rhs As UIntege
) As UInteger
Declare Operator * ( ByRef lhs As LongInt, ByRef rhs As LongInt
As LongInt
Declare Operator * ( ByRef lhs As ULongInt, ByRef rhs As ULongIn
) As ULongInt

Declare Operator * ( ByRef lhs As Single, ByRef rhs As Single )
As Single
Declare Operator * ( ByRef lhs As Double, ByRef rhs As Double )
As Double
```

Usage

```
result = lhs * rhs
```

Parameters

lhs

The left-hand side multiplicand expression.

rhs

The right-hand side multiplicand expression.

Return Value

Returns the product of two multiplicands.

Description

`operator * (Multiply)` returns the product of two multiplicands.

Neither operand is modified in any way.

This operator can be overloaded to accept user-defined types.

Example

```
Dim n As Double  
n = 4 * 5  
Print n  
Sleep
```

Output:

```
20
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- None

See also

- **Mathematical Functions**

Operator / (Divide)



Divides two numeric expressions

Syntax

```
Declare Operator / ( ByRef lhs As Single, ByRef rhs As Single )  
As Single  
Declare Operator / ( ByRef lhs As Double, ByRef rhs As Double )  
As Double
```

Usage

```
result = lhs / rhs
```

Parameters

lhs
The left-hand side dividend expression.

rhs
The right-hand side divisor expression.

Return Value

Returns the quotient of a dividend and divisor.

Description

`operator / (Divide)` returns the quotient of a dividend and divisor.

Neither operand is modified in any way. Unlike with integer division, float division by zero is safe to perform, the quotient will hold a special value representing infinity, converting it to a string returns something like "Inf" or "INF", exact text is platform specific.

This operator can be overloaded to accept user-defined types.

Example

```
Dim n As Double
```

```
Print n / 5
n = 6 / 2.3
Print n
Sleep
```

Output:

```
0
2.608695652173913
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- None

See also

- [Operator \ \(Integer Divide\)](#)
- [Mathematical Functions](#)

Operator \ (Integer Divide)



Divides two `Integer` expressions

Syntax

```
Declare Operator \ ( ByRef lhs As Integer, ByRef rhs As Integer
As Integer
Declare Operator \ ( ByRef lhs As UInteger, ByRef rhs As UInteger
) As UInteger
Declare Operator \ ( ByRef lhs As LongInt, ByRef rhs As LongInt
As LongInt
Declare Operator \ ( ByRef lhs As ULongInt, ByRef rhs As ULongInt
) As ULongInt
```

Usage

```
result = lhs \ rhs
```

Parameters

lhs

The left-hand side dividend expression.

rhs

The right-hand side divisor expression.

Return Value

Returns the quotient of an `Integer` dividend and divisor.

Description

`operator \ (Integer division)` divides two `Integer` expressions and returns the result. Float numeric values are converted to `Integer` by rounding up or down, and the fractional part of the resulting quotient is truncated.

If the divisor (*rhs*) is zero (0), a division by zero error (crash) will be raised.

Neither of the operands are modified in any way.

This operator can be overloaded for user-defined types.

Example

```
Dim n As Double
Print n \ 5
n = 7 \ 2.6 ' ' => 7 \ 3 => 2.33333 => 2
Print n
n = 7 \ 2.4 ' ' => 7 \ 2 => 3.5 => 3
Print n
Sleep
```

Output:

```
0
2
3
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- None

See also

- [Operator / \(Floating-Point Divide\)](#)
- [Operator Mod \(Modulus\)](#)
- [Mathematical Functions](#)

Operator ^ (Exponentiate)



Raises a numeric expression to some power

Syntax

```
Declare Operator ^ ( ByRef lhs As Double, ByRef rhs As Double )  
As Double
```

Usage

```
result = lhs ^ rhs
```

Parameters

lhs

The left-hand side base expression.

rhs

The right-hand side exponent expression.

Return Value

Returns the exponentiation of a base expression raised to some exponent.

Description

operator ^ (Exponentiate) returns the result of a base expression (*lhs*) raised to some exponent expression (*rhs*). ^ works with double float numbers only, operands of other types will be converted into double before performing the exponentiation. Exponent of a fractional value ($1/n$) is the same as taking n th root from the base, for example, ^ (1/3) is the cube root of 2.

Neither of the operands are modified in any way.

Note: this operation is not guaranteed to be fully accurate, and there may be some inaccuracy in the least significant bits of the number. This is particularly noticeable when the result is expected to be an exact number: in these cases, you may find the result is out by a very

small amount. For this reason, you should never assume that an exponentiation expression will be exactly equal to the value you expect.

This also means that you should be wary of using rounding methods such as `Int` and `Fix` on the result: if you expect the result to be an integer value, then there's a chance that it might be slightly lower, and will round down to a value that is one less than you would expect.

This operator can be overloaded for user-defined types.

Note: This operator exists in C/C++ with a different meaning - there it performs a Bitwise `xor`.

Example

```
Dim As Double n
Input "Please enter a positive number: ", n
Print
Print n;" squared is "; n ^ 2
Print "The fifth root of "; n;" is "; n ^ 0.2
Sleep
```

Output:

```
Please enter a positive number: 3.4

3.4 squared is 11.56
The fifth root of 3.4 is 1.27730844458754
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- None

See also

- **Mathematical Functions**

Operator - (Negate)



Changes the sign of a numeric expression

Syntax

```
Declare Operator - ( ByRef rhs As Integer ) As Integer  
Declare Operator - ( ByRef rhs As Single ) As Single  
Declare Operator - ( ByRef rhs As Double ) As Double
```

Usage

```
result = - rhs
```

Parameters

rhs

The right-hand side numeric expression to negate.

Return Value

Returns the negative of the expression.

Description

`operator - (Negate)` is a unary operator that negates the value of its operand.

The operand is not modified in any way.

This operator can be overloaded for user-defined types.

Example

```
Dim n As LongInt  
Print -5  
n = 65432568459  
n = - n  
Print n  
Sleep
```

Output:

```
-5  
-65432568459
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- None

See also

- **Mathematical Functions**

Graphics Keyword List



A list of the keywords and procedures of FreeBASIC's graphics library.

- Add
- Alpha
- And (Graphics Put)
- BLoad
- BSave
- Circle
- Color
- Custom
- Draw
- Draw String
- Event (Message Data From Screenshot)
- Flip
- ImageConvertRow
- ImageCreate
- ImageDestroy
- ImageInfo
- Get (Graphics)
- GetJoystick
- GetMouse
- Inp
- Line (Graphics)
- MultiKey
- Out
- Or (Graphics Put)
- Paint
- Palette
- PReset
- PSet
- Pset (Graphics Put)
- Put (Graphics)
- RGB
- RGBA
- Screen (Graphics)
- ScreenControl
- ScreenCopy
- ScreenEvent
- ScreenGLProc
- ScreenInfo
- ScreenList
- ScreenLock
- ScreenPtr
- ScreenRes
- ScreenSet
- ScreenSync
- ScreenUnlock
- SetMouse
- Trans
- View (Graphics)
- Wait
- Window
- WindowTitle
- Xor (Graphics Put)

- **PCopy**
- **PMap**
- **Point**
- **Pointcoord**

Operators List



List of operators used in FreeBASIC.

Assignment Operators

- `=>` (Assignment)
- `&=` (Concatenate And Assign)
- `+=` (Add And Assign)
- `-=` (Subtract And Assign)
- `*=` (Multiply And Assign)
- `/=` (Divide And Assign)
- `\=` (Integer Divide And Assign)
- `^=` (Exponentiate And Assign)
- `Mod=` (Modulus And Assign)
- `And=` (Conjunction And Assign)
- `Eqv=` (Equivalence And Assign)
- `Imp=` (Implication And Assign)
- `Or=` (Inclusive Disjunction And Assign)
- `Xor=` (Exclusive Disjunction And Assign)
- `Shl=` (Shift Left And Assign)
- `Shr=` (Shift Right And Assign)
- `Let` (Assignment)

Relational Operators

- `=` (Equal)
- `<>` (Not Equal)
- `<` (Less Than)
- `<=` (Less Than Or Equal)
- `>=` (Greater Than Or Equal)
- `>` (Greater Than)

Bitwise Operators

- `And` (Conjunction)
- `Eqv` (Equivalence)
- `Imp` (Implication)
- `Not` (Complement)
- `Or` (Inclusive Disjunction)
- `Xor` (Exclusive Disjunction)

Short Circuit Operators

- `Andalso` (Short Circuit Conjunction)
- `Orelse` (Short Circuit Inclusive Disjunction)

Preprocessor Operators

- `#` (Argument Stringize)
- `##` (Argument Concatenation)

- **Let() (Assignment)**

Type Cast Operators

- **Cast**
- **CPtr**

Arithmetic Operators

- **+ (Add)**
- **- (Subtract)**
- *** (Multiply)**
- **/ (Divide)**
- **\ (Integer Divide)**
- **^ (Exponentiate)**
- **Mod (Modulus)**
- **- (Negate)**
- **Shl (Shift Left)**
- **Shr (Shift Right)**

Indexing Operators

- **() (Array Index)**
- **[] (String Index)**
- **[] (Pointer Index)**

String Operators

- **+ (String Concatenation)**
- **& (String Concatenation With Conversion)**
- **Strptr (String Pointer)**

- **! (Escaped String Literal)**
- **\$ (Non-Escaped String Literal)**

Pointer Operators

- **@ (Address Of)**
- *** (Value Of)**
- **Varptr (Variable Pointer)**
- **Procptr (Procedure Pointer)**

Type or Class Operators

- **. (Member Access)**
- **-> (Pointer To Member Access)**
- **Is (Run-Time Type Information Operator)**

Memory Operators

- **New**
- **Placement New**
- **Delete**

Iteration Operators

- **For, Next, and Step**

Operator () (Array Index)



Returns a reference to an element in an array

Syntax

```
Declare Operator () ( lhs() As T, ByRef rhs As Integer, ... ) By  
As T
```

Usage

```
result = lhs ( rhs [, ...] )
```

Parameters

lhs
An array.

rhs
An index of an element in the array.

T
Any data type.

Description

This operator returns a reference to an element in an array. For multidimensional arrays, multiple indexes must be specified (up to the number of dimensions of the array).

For any dimension *d* in array *a*, any index less than **LBound**(*a*, *d*) or greater than **UBound**(*a*, *d*) will result in a runtime error.

Example

```
Dim array(0 To 4) As Integer = { 0, 1, 2, 3, 4 }  
  
For index As Integer = 0 To 4  
    Print array(index);  
Next
```

```
Print
```

will produce the output:

```
0 1 2 3 4
```

Differences from QB

- None

See also

- `Operator [] (Pointer Index)`

Operator [] (String Index)



Returns a reference to a character in a string

Syntax

```
Declare Operator [] ( ByRef lhs As String, ByRef rhs As Integer
ByRef As UByte
Declare Operator [] ( ByRef lhs As ZString, ByRef rhs As Integer
) ByRef As UByte
Declare Operator [] ( ByRef lhs As WString, ByRef rhs As Integer
) ByRef As T
```

Usage

```
result = lhs [ rhs ]
```

Parameters

lhs

The string (a string reference, not a string returned as local copy).

rhs

A zero-based offset from the first character.

T

The wide-character type (varies per platform).

Description

This operator returns a reference to a specific character in a string:

- This operator must not be used in case of empty string because reference is undefined (inducing runtime error)
- Otherwise, the user must ensure that the index does not exceed the range "[0, Len(*lhs*) - 1]". Outside this range results are undefined.

Example

```
Dim a As String = "Hello, world!"
Dim i As Integer
```

```
For i = 0 To Len(a) - 1
    Print Chr(a[i]) & " ";
Next i
Print
```

Will print

```
H e l l o ,   w o r l d !
```

Differences from QB

- New to FreeBASIC

See also

- **String Operators**

Operator [] (Pointer Index)



Returns a reference to memory offset from an address

Syntax

```
Declare Operator [] ( ByRef lhs As T Pointer, ByRef rhs As Integ  
ByRef As T
```

Usage

```
result = lhs [ rhs ]
```

Parameters

lhs

The base address.

rhs

A signed offset from *lhs*.

T

Any data type.

Description

This operator returns a reference to a value some distance in memory from a base address. It is essentially shorthand for " $*(lhs + rhs)$ "; but exactly the same thing. Like pointer arithmetic, any type of **Pointer** can be indexed except for an **Any Pointer**. Also, like pointer arithmetic, it is up to the user to make sure meaningful data is being accessed.

When indexing a '2-dimensional' pointer (i.e. a $T \text{ Ptr Ptr}$), the first (leftmost) index is applied before the second: For example, $Pt[I1][I2]$ is equivalent to $(Pt[I1] + I2) = (*(Pt + I1) + I2)$

In general, when using an '*n*-dimensional' pointer: $Pt[I1][I2] \dots [Ir]$, the index order (from left to right) corresponds to the dereferencing order.

This operator can be overloaded for user-defined types.

Example

```
' ' initialize a 5-element array
Dim array(4) As Integer = { 0, 1, 2, 3, 4 }

' ' point to the first element
Dim p As Integer Ptr = @array(0)

' ' use pointer indexing to output array elements
For index As Integer = 0 To 4
    Print p[index];
Next
Print
```

Will give the output,

```
0 1 2 3 4
```

Differences from QB

- New to FreeBASIC

See also

- **Pointer Arithmetic**
- **Operator * (Value Of)**
- **Operator [] (String Index)**
- **Operator () (Array Index)**
- **Operator + (Add)**
- **Operator - (Subtract)**
- **Pointer Operators**

Operator + (String Concatenation)



Concatenates two strings

Syntax

```
Declare Operator + ( ByRef lhs As String, ByRef rhs As String )  
As String  
Declare Operator + ( ByRef lhs As ZString, ByRef rhs As ZString  
As ZString  
Declare Operator + ( ByRef lhs As WString, ByRef rhs As WString  
As WString
```

Usage

```
result = lhs + rhs
```

Parameters

lhs

The left-hand side string to concatenate.

rhs

The right-hand side string to concatenate.

Description

This operator concatenates two strings. Unlike **Operator & (String Concatenation With Conversion)** both expressions *must* be strings, and may not be converted (in fact, any attempt to concatenate a string with a non-string or two non-strings will result in a type mismatch error with the exception of when operator overloading is used in a UDT).

Example

```
Dim As String a = "Hello, ", b = "World!"  
Dim As String c  
c = a + b  
Print c
```

Output:

```
Hello, World!
```

Differences from QB

- None

See also

- **Operator + (Add)**
- **Operator & (String Concatenation With Conversion)**
- **str**

Operator & (String Concatenation With Conversion)



Concatenates two strings, converting non-strings to strings as needed

Syntax

```
Declare Operator & ( ByRef lhs As T, ByRef rhs As U ) As V
```

Usage

```
result = lhs & rhs
```

Parameters

lhs

The left-hand side expression to concatenate.

T

Any standard data type or user-defined type that can be converted to standard data type.

rhs

The right-hand side expression to concatenate.

U

Any standard data type or user-defined type that can be converted to standard data type.

V

The resultant string type (varies with operands).

Description

This operator concatenates two expressions. If either of the expressions is not a string type, it is converted to **String** with **Str**.

If either of the expressions is a **wString**, a **wString** is returned, otherwise a **String** is returned.

Note: This operator exists in C/C++ with a different meaning - there it performs a bitwise **And**.

Example

```
Dim As String A,C
Dim As Single B
A="The result is: "
B=124.3
C=A & B
Print C
Sleep
```

Output:

```
The result is: 124.3
```

Differences from QB

- New to FreeBASIC

See also

- **Operator + (String Concatenation)**
- **str**

Operator = (Equal)



Compares two expressions for equality

Syntax

```
Declare Operator = ( ByRef lhs As Byte, ByRef rhs As Byte ) As Integer
Declare Operator = ( ByRef lhs As UByte, ByRef rhs As UByte ) As Integer
Declare Operator = ( ByRef lhs As Short, ByRef rhs As Short ) As Integer
Declare Operator = ( ByRef lhs As UShort, ByRef rhs As UShort ) As Integer
Declare Operator = ( ByRef lhs As Integer, ByRef rhs As Integer ) As Integer
Declare Operator = ( ByRef lhs As UInteger, ByRef rhs As UInteger ) As Integer
Declare Operator = ( ByRef lhs As LongInt, ByRef rhs As LongInt ) As Integer
Declare Operator = ( ByRef lhs As ULongInt, ByRef rhs As ULongInt ) As Integer

Declare Operator = ( ByRef lhs As Single, ByRef rhs As Single ) As Integer
Declare Operator = ( ByRef lhs As Double, ByRef rhs As Double ) As Integer

Declare Operator = ( ByRef lhs As String, ByRef rhs As String ) As Integer
Declare Operator = ( ByRef lhs As ZString, ByRef rhs As ZString ) As Integer
Declare Operator = ( ByRef lhs As WString, ByRef rhs As WString ) As Integer

Declare Operator = ( ByRef lhs As T, ByRef rhs As T ) As Integer

Declare Operator = ( ByRef lhs As Boolean, ByRef rhs As Boolean ) As Integer
```

Usage

```
result = lhs = rhs
```

Parameters

lhs

The left-hand side expression to compare to.

rhs

The right-hand side expression to compare to.

T

Any pointer type.

Return Value

Returns negative one (-1) if expressions are equal, or zero (0) if unequal.

Description

`operator = (Equality)` is a binary operator that compares two expressions mainly in the form of an `Integer`: negative one (-1) for true and zero (0) for false. If the operands are both `Boolean`, the return type is also `Boolean`. The arguments

This operator can be overloaded to accept user-defined types as well.

`operator = (Equality)` should not be confused with initializations or assignments.

Example

```
Dim i As Integer = 0      '' initialization: initial
i = 420                  '' assignment: assign to i

If (i = 69) Then        '' equation: compare the e
    Print "serious error: i should equal 420"
    End -1
End If
```

`operator <>` (Inequality) is complement to `operator = (Equality)`, and `operator Not` (Bit-wise Complement).

```
If (420 = 420) Then Print "(420 = 420) is true."
If Not (69 <> 69) Then Print "not (69 <> 69) is
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- none

See also

- `operator <>` (Inequality)
- `operator = [>]` (Assignment)

Operator <> (Not Equal)



Compares two expressions for inequality

Syntax

```
Declare Operator <> ( ByRef lhs As Byte, ByRef rhs As Byte ) As  
Declare Operator <> ( ByRef lhs As UByte, ByRef rhs As UByte ) As  
Declare Operator <> ( ByRef lhs As Short, ByRef rhs As Short ) As  
Declare Operator <> ( ByRef lhs As UShort, ByRef rhs As UShort ) As  
Declare Operator <> ( ByRef lhs As Integer, ByRef rhs As Integer ) As  
Declare Operator <> ( ByRef lhs As UInteger, ByRef rhs As UInteger ) As  
Declare Operator <> ( ByRef lhs As LongInt, ByRef rhs As LongInt ) As  
Declare Operator <> ( ByRef lhs As ULongInt, ByRef rhs As ULongInt ) As  
  
Declare Operator <> ( ByRef lhs As Single, ByRef rhs As Single ) As  
Declare Operator <> ( ByRef lhs As Double, ByRef rhs As Double ) As  
  
Declare Operator <> ( ByRef lhs As String, ByRef rhs As String ) As  
Declare Operator <> ( ByRef lhs As ZString, ByRef rhs As ZString ) As  
Declare Operator <> ( ByRef lhs As WString, ByRef rhs As WString ) As  
  
Declare Operator <> ( ByRef lhs As T, ByRef rhs As T ) As Integer  
  
Declare Operator <> ( ByRef lhs As Boolean, ByRef rhs As Boolean ) As Integer
```

Usage

```
result = lhs <> rhs
```

Parameters

lhs

The left-hand side expression to compare to.

rhs

The right-hand side expression to compare to.

T

Any pointer type.

Return Value

Returns negative one (-1) if expressions are not equal, or zero (0) if e

Description

operator <> (Not equal) is a binary operator that compares two expressions for inequality and returns the result - a boolean value mainly in the form of a negative one (-1) for true and zero (0) for false. Only if the left and right types are both **Boolean**, the return type is also **Boolean**. The arguments can be in any way.

This operator can be overloaded to accept user-defined types as well.

Example

```
Dim As String a = "hello", b = "world"
Dim As Integer i = 10, j = i

If (a <> b) Then
    Print a & " does not equal " & b
End If

If (i <> j) Then
    Print "error: " & i & " does not equal " & j
End If
```

operator = (Equal) is complement to **operator <> (Not equal)**, and is identical when combined with **operator Not (Bit-wise Complement)**.

```
If (69 <> 420) Then Print "(69 <> 420) is true."
If Not (69 = 420) Then Print "not (69 = 420) is true."
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- none

See also

- `operator =` (Equal)

Operator < (Less Than)



Compares an expression less than another expression

Syntax

```
Declare Operator < ( ByRef lhs As Byte, ByRef rhs As Byte ) As Integer
Declare Operator < ( ByRef lhs As UByte, ByRef rhs As UByte ) As Integer
Declare Operator < ( ByRef lhs As Short, ByRef rhs As Short ) As Integer
Declare Operator < ( ByRef lhs As UShort, ByRef rhs As UShort ) As Integer
Declare Operator < ( ByRef lhs As Integer, ByRef rhs As Integer ) As Integer
Declare Operator < ( ByRef lhs As UInteger, ByRef rhs As UInteger ) As Integer
Declare Operator < ( ByRef lhs As LongInt, ByRef rhs As LongInt ) As Integer
Declare Operator < ( ByRef lhs As ULongInt, ByRef rhs As ULongInt ) As Integer

Declare Operator < ( ByRef lhs As Single, ByRef rhs As Single ) As Integer
Declare Operator < ( ByRef lhs As Double, ByRef rhs As Double ) As Integer

Declare Operator < ( ByRef lhs As String, ByRef rhs As String ) As Integer
Declare Operator < ( ByRef lhs As ZString, ByRef rhs As ZString ) As Integer
Declare Operator < ( ByRef lhs As WString, ByRef rhs As WString ) As Integer

Declare Operator < ( ByRef lhs As T, ByRef rhs As T ) As Integer
```

Usage

```
result = lhs < rhs
```

Parameters

lhs

The left-hand side expression to compare to.

rhs

The right-hand side expression to compare to.

T

Any pointer type.

Return Value

Returns negative one (-1) if the left-hand side expression is less than expression, or zero (0) if greater than or equal.

Description

operator < (Less than) is a binary operator that compares two expressions and returns the result - a boolean value in the form of an **Integer**: negative for true and zero (0) for false. The arguments are not modified in any way.

This operator can be overloaded to accept user-defined types as well.

Example

```
Const size As Integer = 4
Dim array(size - 1) As Integer = { 1, 2, 3, 4 }

Dim index As Integer = 0
While (index < size)
    Print array(index)
    index += 1
Wend
```

operator >= (Greater than or equal) is complement to **operator < (Less than)** and is functionally identical when combined with **operator Not** (Bit-wise Complement).

```
If (69 < 420) Then Print "(69 < 420) is true."
If Not (69 >= 420) Then Print "not (69 >= 420)"
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- none

See also

- **operator >=** (Greater than or equal)

Operator <= (Less Than Or Equal)



Compares an expression less than or equal to another expression

Syntax

```
Declare Operator <= ( ByRef lhs As Byte, ByRef rhs As Byte ) As  
Declare Operator <= ( ByRef lhs As UByte, ByRef rhs As UByte ) As  
Declare Operator <= ( ByRef lhs As Short, ByRef rhs As Short ) As  
Declare Operator <= ( ByRef lhs As UShort, ByRef rhs As UShort ) As  
Declare Operator <= ( ByRef lhs As Integer, ByRef rhs As Integer ) As  
Declare Operator <= ( ByRef lhs As UInteger, ByRef rhs As UInteger ) As  
Declare Operator <= ( ByRef lhs As LongInt, ByRef rhs As LongInt ) As  
Declare Operator <= ( ByRef lhs As ULongInt, ByRef rhs As ULongInt ) As  
  
Declare Operator <= ( ByRef lhs As Single, ByRef rhs As Single ) As  
Declare Operator <= ( ByRef lhs As Double, ByRef rhs As Double ) As  
  
Declare Operator <= ( ByRef lhs As String, ByRef rhs As String ) As  
Declare Operator <= ( ByRef lhs As ZString, ByRef rhs As ZString ) As  
Declare Operator <= ( ByRef lhs As WString, ByRef rhs As WString ) As  
  
Declare Operator <= ( ByRef lhs As T, ByRef rhs As T ) As Integer
```

Usage

```
result = lhs <= rhs
```

Parameters

lhs

The left-hand side expression to compare to.

rhs

The right-hand side expression to compare to.

T

Any pointer type.

Return Value

Returns negative one (-1) if the left-hand side expression is less than right-hand side expression, or zero (0) if greater than.

Description

`operator <=` (**Less than or Equal**) is a binary operator that compares less than or equal to another expression and returns the result - a boolean form of an **Integer**: negative one (-1) for true and zero (0) for false. This is not modified in any way.

This operator can be overloaded to accept user-defined types as well.

Example

`operator >` (Greater than) is complement to `operator <=` (**Less than or Equal**) functionally identical when combined with `operator Not` (Bit-wise Complement).

```
If (69 <= 420) Then Print "(69 <= 420) is true."
If Not (60 > 420) Then Print "not (420 > 69) is true."
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- none

See also

- `operator >` (Greater than)

Operator >= (Greater Than Or Equal)



Compares an expression greater than or equal to another expression

Syntax

```
Declare Operator >= ( ByRef lhs As Byte, ByRef rhs As Byte ) As  
Declare Operator >= ( ByRef lhs As UByte, ByRef rhs As UByte ) A  
Declare Operator >= ( ByRef lhs As Short, ByRef rhs As Short ) A  
Declare Operator >= ( ByRef lhs As UShort, ByRef rhs As UShort )  
Declare Operator >= ( ByRef lhs As Integer, ByRef rhs As Integer  
Declare Operator >= ( ByRef lhs As UInteger, ByRef rhs As UInteg  
Declare Operator >= ( ByRef lhs As LongInt, ByRef rhs As LongInt  
Declare Operator >= ( ByRef lhs As ULongInt, ByRef rhs As ULongI  
  
Declare Operator >= ( ByRef lhs As Single, ByRef rhs As Single )  
Declare Operator >= ( ByRef lhs As Double, ByRef rhs As Double )  
  
Declare Operator >= ( ByRef lhs As String, ByRef rhs As String )  
Declare Operator >= ( ByRef lhs As ZString, ByRef rhs As ZString  
Declare Operator >= ( ByRef lhs As WString, ByRef rhs As WString  
  
Declare Operator >= ( ByRef lhs As T, ByRef rhs As T ) As Intege
```

Usage

```
result = lhs >= rhs
```

Parameters

lhs

The left-hand side expression to compare to.

rhs

The right-hand side expression to compare to.

T

Any pointer type.

Return Value

Returns negative one (-1) if the left-hand side expression is greater than the right-hand side expression, or zero (0) if less than.

Description

`operator >=` (**Greater than or Equal**) is a binary operator that compares greater than or equal to another expression and returns the result - in the form of an **Integer**: negative one (-1) for true and zero (0) for false are not modified in any way.

This operator can be overloaded to accept user-defined types as well.

Example

`operator <` (Less than) is complement to `operator >=` (**Greater than or Equal**) functionally identical when combined with `operator Not` (Bit-wise Com

```
If (420 >= 69) Then Print "(420 >= 69) is true.  
If Not (420 < 69) Then Print "not (420 < 69) is
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- none

See also

- `operator <` (Less than)

Operator > (Greater Than)



Compares an expression greater than another expression

Syntax

```
Declare Operator > ( ByRef lhs As Byte, ByRef rhs As Byte ) As Integer
Declare Operator > ( ByRef lhs As UByte, ByRef rhs As UByte ) As Integer
Declare Operator > ( ByRef lhs As Short, ByRef rhs As Short ) As Integer
Declare Operator > ( ByRef lhs As UShort, ByRef rhs As UShort ) As Integer
Declare Operator > ( ByRef lhs As Integer, ByRef rhs As Integer ) As Integer
Declare Operator > ( ByRef lhs As UInteger, ByRef rhs As UInteger ) As Integer
Declare Operator > ( ByRef lhs As LongInt, ByRef rhs As LongInt ) As Integer
Declare Operator > ( ByRef lhs As ULongInt, ByRef rhs As ULongInt ) As Integer

Declare Operator > ( ByRef lhs As Single, ByRef rhs As Single ) As Integer
Declare Operator > ( ByRef lhs As Double, ByRef rhs As Double ) As Integer

Declare Operator > ( ByRef lhs As String, ByRef rhs As String ) As Integer
Declare Operator > ( ByRef lhs As ZString, ByRef rhs As ZString ) As Integer
Declare Operator > ( ByRef lhs As WString, ByRef rhs As WString ) As Integer

Declare Operator > ( ByRef lhs As T, ByRef rhs As T ) As Integer
```

Usage

```
result = lhs > rhs
```

Parameters

lhs

The left-hand side expression to compare to.

rhs

The right-hand side expression to compare to.

T

Any pointer type.

Return Value

Returns negative one (-1) if the left-hand side expression is greater than the right-hand side expression, or zero (0) if less than or equal.

Description

operator > (Greater than) is a binary operator that compares an expression with another expression and returns the result - a boolean value in the form of a negative one (-1) for true and zero (0) for false. The arguments are not in any particular way.

This operator can be overloaded to accept user-defined types as well.

Example

operator <= (Less than or equal) is complementary to **operator >** (Greater than). It is functionally identical when combined with **operator Not** (Bit-wise Complement).

```
If (420 > 69) Then Print "(420 > 69) is true."
If Not (420 <= 69) Then Print "not (420 <= 69)"
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- none

See also

- **operator <=** (Less than or equal)

Operator # (Preprocessor Stringize)



Preprocessor operator to convert macro arguments to strings

Syntax

`#macro_argument`

Description

This operator converts the *macro_argument* into a string whose value is the name of the argument. This substitution is made during the macro expansion, previous to compilation.

Note: because of this feature, care should be taken when using file-handling statements in a macro. Because of potential ambiguity with file-handling statements that take a "#filename" parameter, if filename is one of the macro parameters, it may be necessary to wrap the filename expression in parenthesis (e.g. "#(filename)"), to separate it from the # sign. Otherwise, filename will be stringized in the macro.

Example

```
#define SEE(x) Print #x ;" = "; x
Dim variable As Integer, another_one As Integer
variable=1
another_one=2
SEE(variable)
SEE(another_one)
```

Output:

```
variable = 1
another_one = 2
```

Differences from QB

- New to FreeBASIC

See also

- [Preprocessor](#)

Operator ## (Preprocessor Concatenate)



Preprocessor operator to concatenate strings

Syntax

```
text##text
```

Description

This operator creates a new token by concatenating the texts at both sides of it. This text can be recognized by other macros and further expanded. One use, is to create a macro that expands to different macro names, variable names, and function names depending on the arguments received.

Example

```
#define Concat(t,n) t##n

Print concat (12,34)

Dim Concat (hello,world) As Integer
Concat (hello,world)=99
Print helloworld
```

Output:

```
1234
99
```

Differences from QB

- New to FreeBASIC

See also

- **Preprocessor**

Operator ! (Escaped String Literal)



Explicitly indicates that a string literal should be processed for escape sequences.

Syntax

```
!"text"
```

Parameters

!

The preprocessor escaped string operator

"text"

The string literal containing escape characters

Description

This operator explicitly indicates that the string literal following it (wrapped in double quotes) should be processed for escape sequences. This a preprocessor operator and can only be used with string literals at compile time.

The default behavior for string literals is that they not be processed for escape sequences. **Option Escape** can be used in the *-lang fblite* dialect to override this default behaviour causing all strings to be processed for escape sequences.

Use the **\$ Operator (Non-Escaped String Literal)** operator to explicitly indicate that a string should not be processed for escape sequences.

Example

```
Print "Some escape sequence examples:"
Print !"1.\t\tsingle quote (\\\' ) : \'
Print !"2.\t\tdouble quote (\\\" ) : \"
Print !"3.\t\tbackslash (\\\\ ) : \\
Print !"4.\t\tascii char (\\65): \65"
```

```
' ' OUTPUT:
' '
' ' Some escape sequence examples:
' ' 1. single quote (\') : '
' ' 2. double quote (\") : "
' ' 3. backslash (\\) : \
' ' 4. ascii char (\65): A
```

Differences from QB

- New to FreeBASIC

See also

- **Operator \$ (Non-Escaped String Literal)**
- **Option Escape**
- **Preprocessor**
- **Literals**
- **Escape Sequences**

Operator \$ (Non-Escaped String Literal)



Explicitly indicates that a string literal should not be processed for escape sequences.

Syntax

```
$"text"
```

Parameters

\$

The preprocessor non-escaped operator

"text"

The string literal

Description

This operator explicitly indicates that the string literal following it (wrapped in double quotes) should not be processed for escape sequences. This a preprocessor operator and can only be used with string literals at compile time.

The default behavior for string literals is that they not be processed for escape sequences. However, **option Escape** in the *-lang fblite* dialect can be used to override this default behaviour causing all strings to be processed for escape sequences.

Use the **! Operator (Escaped String Literal)** to explicitly indicate that a string should be processed for escape sequences.

Example

```
' ' Compile with -lang fblite or qb
#lang "fblite"
Print "Default"
Print "Backslash  : \\"
```


Operator @ (Address Of)



Returns the address of a string literal, variable, object or procedure

Syntax

```
Declare Operator @ ( ByRef rhs As T ) As T Pointer
```

Usage

```
result = @ rhs
```

Parameters

rhs

The string literal, variable, object or procedure to retrieve the address
T

Any **standard**, **user-defined** or procedure type.

Return Value

Returns the address of the right-hand side (*rhs*) operand.

Description

operator @ (Address of) returns the memory address of its operand.

When the operand is of type **String**, the address of the internal string (pointer) to retrieve the address of the string data.

The operand cannot be an array, but may be an array element. For example, `"myarray(0)"`.

This operator can be overloaded for user-defined types.

Example

```
'This program demonstrates the use of the @ operator  
Dim a As Integer
```

```
Dim b As Integer
```

```
Dim addr As Integer Ptr
```

```
a = 5      'Here we place the values 5 and 10 into a  
b = 10
```

```
'Here, we print the value of the variables, then w  
Print "The value in A is ";a;" but the pointer to  
Print "The value in B is ";b;" but the pointer to
```

```
'Now, we will take the integer ptr above, and use  
'Note that the * will check the value in the ptr,  
'for a normal variable.
```

```
addr = @a
```

```
Print "The pointer addr is now pointing at the mem
```

```
addr = @b
```

```
Print "The pointer addr is now pointing at the mem
```

```
'This program demonstrates how the @ symbol can be  
'to create pointers to subroutines.
```

```
Declare Sub mySubroutine ()
```

```
Dim say_Hello As Sub()
```

```
say_Hello = @mySubroutine      'We tell say_Hello to  
'The sub() datatype ac
```

```
say_Hello() 'Now we can run say_Hello just like my
```

```
Sub mySubroutine
```

```
Print "hi"  
End Sub
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- New to FreeBASIC

See also

- [Operator * \(Value Of\)](#)
- [Pointers](#)

Operator * (Value Of)



Dereferences a pointer

Syntax

```
Declare Operator * ( ByRef rhs As T Pointer ) ByRef As T
```

Usage

```
result = * rhs
```

Parameters

rhs

The address to dereference.

T

Any **standard**, **user-defined** or procedure type.

Return Value

Returns a reference to the value stored at the address *rhs*.

Description

operator * (Value of) returns a reference to the value stored at an address. The operand is not modified in any way.

As a reference, the result of this operator can be used on the left-hand side of an assignment.

This operator can be overloaded for user-defined types.

Example

```
'This program demonstrates the use of * to utilize  
Dim a As Integer  
Dim pa As Integer Ptr  
  
pa=@a 'Here, we use the @ operator to point our ir
```

```
' 'a' is, in this case, a standard integer variabl  
a=9      'Here we give 'a' a value of 9.  
  
Print "The value of 'a' is";*pa 'Here, we display  
  
*pa = 1 'Here we use our pointer to change the val  
Print "The new value of 'a' is";a 'Here we display
```

Output:

```
The value of 'a' is 9  
The new value of 'a' is 1
```

Dialect Differences

- In the *-lang qb* dialect, this operator cannot be overloaded.

Differences from QB

- New to FreeBASIC

See also

- [Operator @ \(Address Of\)](#)
- [Operator \[\] \(Pointer Index\)](#)
- [Pointers](#)

Operator . (Member Access)



Returns a reference to a member from a reference to an object

Syntax

```
Declare Operator . ( ByRef lhs As T ) ByRef As U
```

Usage

```
result = lhs . rhs
```

Parameters

lhs

An object.

T

A user-defined type.

rhs

The name of a member to access.

U

The type that *rhs* refers to.

Return Value

Returns a reference to the member specified by *rhs*.

Description

`operator . (Member access)` returns a reference to a member of an object.

`operator . (Member access)` can also be used to access members of an implicit object inside a `With..End With` block.

This operator cannot be overloaded.

Example

```
Type T
  As Integer a, b
End Type

Dim x As T

'' Access the member 'a' of x.
x.a = 10

'' Access the member 'b' of x.
With x
  .b = 20
End With
```

Dialect Differences

- None

Differences from QB

- None

See also

- Operator -> (Pointer To Member Access)
- Operator @ (Address Of)
- Operator * (Value Of)
- With..End With

Operator -> (Pointer To Member Access)



Returns a reference to a member from a pointer to an object

Syntax

```
Declare Operator -> ( ByRef lhs As T Ptr ) ByRef As U
```

Usage

```
result = lhs -> rhs
```

Parameters

lhs

The address of an object.

T

A user-defined type.

rhs

The name of a member to access.

U

The type that *rhs* refers to.

Return Value

Returns a reference to the member specified by *rhs*.

Description

`operator -> (Pointer to member access)` returns a reference to a member of an object through a pointer to that object. It has the effect of dereferencing a pointer to an object, then using `operator . (Member Access)`. For example, "*p->member*" is equivalent to "*x.member*", if *x* is an object of user-defined type and *p* is a pointer to an object of the same type.

This operator can be overloaded for user-defined types.

Example

```
Type rect
    x As Integer
    y As Integer
End Type

Dim r As rect
Dim rp As rect Pointer = @r

rp->x = 4
rp->y = 2

Print "x = " & rp->x & ", y = " & rp->y
Sleep
```

Dialect Differences

- Not available in the *-lang qb* dialect.

Differences from QB

- New to FreeBASIC

See also

- **Operator . (Member Access)**
- **Operator @ (Address Of)**
- **Operator * (Value Of)**

Operator For (Iteration)



Declares or defines operators used by a **For . . . Next** loop with user defined type variables

Syntax

```
{ Type | Class | Union } typename  
Declare Operator For ()  
Declare Operator For ( [ ByRef | ByVal ] stp As typename )  
...  
End { Type | Class | Union }
```

Usage

```
For iterator [ As typename ] = start_value To end_value [ Step  
step_value ]  
[ ...statements... ]  
Next
```

Parameters

typename
name of the **Type**, **Class**, or **Union**
stp, *step_value*
a *typename* object used as an incremental value
iterator
a *typename* object used as an iterator
end_value
a *typename* object used as a loop-terminating value
start_value
a *typename* object used to copy construct or assign to the iterator initially

Description

Operator For, **Operator Next** and **Operator Step** can be overloaded in user-defined type definitions to allow objects of that type to be used as iterators and step values in **For . . . Next** loops.

Operator For is called immediately after copy constructing or assigning to the iterator object, and allows the object to perform any additional

initialization needed in preparation for the loop.

The first version of **operator For** is used if no step value is given in the **For...Next** statement. If a step value is given, the second version is used and is passed the step value.

Example

See the **Operator Step** examples.

Dialect Differences

- Only available in the *-lang fb* dialect.

See also

- **Operator Next**
- **Operator Step**
- **For...Next**

Variable Declarations



Statements to declare and allocate space for variables.

Dim

Declares a variable at the current scope.

Const

Declares a non-modifiable variable.

Scope

Begins a new scope block.

Static

Declares variables in a procedure that retain their value between calls.

Shared

Used with **Dim** allows variables to be visible throughout a module.

Var

Declares variables where the data type is implied from an initializer.

Declaration

- Enum...End Enum
- Type...End Type
- Union...End Union

- Extends
- Field
- Object

Referencing

- Temporary Types
- This
- Base (Member Access)
- Type Alias
- With

Member Procedures

- Base (Initialization)
- Constructor
- Destructor
- Function
- Operator
- Override
- Property
- Sub
- Static (Member)
- Virtual
- Abstract
- Const (Member)

Member Access Control

- Public: (Access Control)
- Private: (Access Control)
- Protected: (Access Control)

Standard Data Types



Built-in data types

Integer types

Types that store integer values, whose range is determined by the size of the data type and its signedness.

Floating-point types

Types that store real number values, whose range and precision is determined by the size of the data type.

Boolean types

Types that store boolean values.

Data Type Modifiers

Specifies additional characteristics of a standard or user-defined data type.

String types

Types that store or point to an array of characters.

Class types

Types that provide special capabilities to be used directly or to be extended by user-defined types

Integer types

Byte and UByte

8-bit wide data types that store integer values.

Short and UShort

16-bit wide data types that store integer values.

Long and ULong

32-bit wide data types that store integer values.

Integer and UInteger

32-bit or 64-bit wide data types that store integer values.

LongInt and ULongInt

64-bit wide data types that store integer values.

Data Type Modifiers

Const

Specifies a read only type.

Pointer and Ptr

Modifies types to be pointer types.

Unsigned

Specifies an unsigned integer type.

String types

String

Fixed-length and variable-length strings with built-in memory management.

ZString

Floating-point types

Single

32-bit wide data types that store real number values.

Double

64-bit wide data types that store real number values.

Boolean types

Boolean

1-bit wide data types that store boolean values.

Fixed-length and variable-length null-terminated strings.

WString

Fixed-length and variable-length null-terminated strings of wide characters.

Class types

Object

Super class providing run-time type information

See also

- **Variable types and limits**

Standard Data Type Limits



Standard variable types and limits.

Numeric Types

Type	Size in bits	Format	Minimum Value	Maximum Value
BYTE	8	signed integer	-128	+127
UBYTE	8	unsigned integer	0	+255
SHORT	16	signed integer	-32768	+32767
USHORT	16	unsigned integer	0	65535
LONG	32	signed integer	-2147483648	+2147483647
ULONG	32	unsigned integer	0	+4294967295
INTEGER	32/64 [*]	signed integer	[*]32bit: -2147483648, 64bit: -9223372036854775808	[*]32bit: +2147483647, 64bi +9223372036854775807
UINTeger	32/64 [*]	unsigned integer	0	[*]32bit: +4294967295, 64bi +18446744073709551615
LONGINT	64	signed integer	-9223372036854775808	+9223372036854775807
ULONGINT	64	unsigned integer	0	+18446744073709551615
SINGLE	32	floating point	[**]+/-1.401 298 E-45	[**]+/-3.402 823 E+38
DOUBLE	64	floating point	[**]+/-4.940 656 458 412 465 E- 324	[**]+/-1.797 693 134 862 31 E+308
enums	32/64 [*]	signed integer	[*]32bit: -2147483648, 64bit: -9223372036854775808	[*]32bit: +2147483647, 64bi +9223372036854775807

[*] *Integer* and *UInteger* data types vary with platform, matching the size
[**] The minimum and maximum values for the floating-point types *Single*
respectively, the values closest to zero and the values closest to positive

infinity.

String Types

Type	Character Size (in bytes)	Minimum Size (in characters)	Maximum Size (in characters)
String	1	0	[**]32bit: +2147483647, 64bit: +9223372036854775807
Zstring	1	0	[**]32bit: +2147483647, 64bit: +9223372036854775807
Wstring	[*]	[*]0	[*, **]32bit: +2147483647, 64bit: +9223372036854775807

[*] *Unicode, or "wide", characters vary in both size and availability with p*
[**] *All runtime library string procedures take and produce **Integer** value: positions. The actual maximum size will vary (smaller) with storage local*

Arrays

Platform	Maximum Subscript Range	Maximum Elements per Dimension	Minimum/Maximum Dimensions	Ma
32bit	[*][-2147483648, +2147483647]	[*]+2147483647	1/9	[*]+
64bit	[*][-9223372036854775808, +9223372036854775807]	[*]+9223372036854775807	1/9	[*]+

[*] *All runtime library array procedures take and produce **Integer** values indexes. The actual limits will vary (smaller) with the number of dimension storage location and/or platform.*

See also

ProPgIdentifierRules usage of suffixes for variables
ProPgLiterals usage of suffixes for literals / numbers

Converting Data Types



Operators and procedures that convert between different types.

Generic conversions

Operators to convert between arbitrary types.

Conversions to integral types

Operators to convert to integral types.

Conversions to floating-point types

Operators to convert to floating-point types.

Conversions to/from string types

Operators to convert to and from string types.

Conversion to boolean types

Operators to convert to boolean types.

Generic conversions

Cast and CPtr

Converts expressions between different types.

Conversions to integral types

CByte and CByte

Converts numeric expressions to 8-bit values.

CShort and CShort

Converts numeric expressions to 16-bit values.

CLng and CULng

Converts numeric expressions to 32-bit values.

CInt and CUInt

Converts numeric expressions to 32-bit or 64-bit values.

CLngInt and CULngInt

Converts numeric expressions to 64-bit values.

CSign

Conversions to floating-point types

CSng and CDbl

Converts a numeric or string expression to floating-point values.

Conversions to/from string types

Str and WStr

Converts numeric expressions or booleans to their string representation.

Val

Converts a numeric string expression to a floating-point value.

ValInt and ValUInt

Converts numeric string expressions to integer values.

ValLng and ValULng

Converts numeric string

Converts a numeric expression to a signed-type value.

CUnsg

Converts a numeric expression to an unsigned-type value.

expressions to long values.

Conversion to boolean types

Cbool

Converts a numeric or string expression to a boolean value.

Procedures that operate on one or more operands.

FreeBASIC has numerous *operators* that perform a certain function with their *operands*. Many operators use a "operand *operator* operand" syntax, like **operator = (Assignment)** or **operator +**, while others are called like normal procedures, like **operator Strptr**.

Assignment operators

Operators which assign the value of one operand to the other.

Arithmetic operators

Operators that perform mathematical computations on their operands and return the result.

Conditional operators

Operators that compare the relationship between their operands.

Logical operators

Operators that perform bitwise computations with their operands and return the result.

Short circuit operators

Operators that perform short circuit evaluations with their operands and return the result.

Indexing operators

Operators that return references to variables or objects based on an index value.

String operators

Operators overloaded to work with strings.

Preprocessor operators

Operators that control preprocessor behavior.

Pointer operators

Operators that work with pointers and addresses.

Type or Class operators

Operators that provide access to **Type** or **Class** members.

Memory operators

Operators that allocate memory for and construct objects.

Iterating operators

Operators that use iterator objects in **For . . . Next** statements.

Assignment Operators



Operators that assign values to operands

The assignment operators perform an assignment to the first, or *left-hand side*, operand based on the value of the second, or *right-hand side*, operand. Most of the assignment operators are combination operators, i.e., that they first perform a mathematical or bitwise operation on the two operands, then assign the result to the *left-hand side* operand.

Operator [=] (Assignment)

Assigns the value of one operand to the other.

Operator &= (Concatenate And Assign)

Assigns the value of a concatenation between two operands.

Operator += (Add And Assign)

Assigns the value of an addition between two operands.

Operator -= (Subtract And Assign)

Assigns the value of a subtraction between two operands.

Operator *= (Multiply And Assign)

Assigns the value of a multiplication between two operands.

Operator /= (Divide And Assign)

Assigns the value of a division between two operands.

Operator \= (Integer Divide And Assign)

Assigns the value of an integer

Operator Mod= (Modulus And Assign)

Assigns the value of a modulus between two operands.

Operator And= (Conjunction And Assign)

Assigns the value of a bitwise conjunction between two operands.

Operator Eqv= (Equivalence And Assign)

Assigns the value of a bitwise equivalence between two operands.

Operator Imp= (Implication And Assign)

Assigns the value of a bitwise implication between two operands.

Operator Or= (Inclusive Disjunction And Assign)

Assigns the value of a bitwise inclusive or between two operands.

Operator Xor= (Exclusive Disjunction And Assign)

Assigns the value of a bitwise

divide between two operands.

Operator ^= (Exponentiate And Assign)

Assigns the value of a exponentiation between two operands.

Operator Let (Assignment)

Assigns the value of one user defined type to another.

Operator Let() (Assignment)

Assigns the fields of a user defined type to a list of variables.

exclusive or between two operands.

Operator Shl= (Shift Left And Assign)

Assigns the value of a bitwise shift left of an operand.

Operator Shr= (Shift Right And Assign)

Assigns the value of a bitwise shift right of an operand.

Arithmetic Operators



Operators that can be used in mathematical expressions

The mathematical operators perform mathematical operations with the values of their operands and return the results.

Operator + (Add)

Returns the result of an addition of two operands.

Operator - (Subtract)

Returns the result of a subtraction of two operands.

Operator * (Multiply)

Returns the result of a multiplication of two operands.

Operator / (Divide)

Returns the result of a division of two operands.

Operator \ (Integer Divide)

Returns the result of an integer divide of two operands.

Operator ^ (Exponentiate)

Returns the result of an exponentiation of two operands.

Operator Mod (Modulus)

Returns the result of a modulus of two operands.

Operator - (Negate)

Returns the result of a negation of an operand.

Operator Shl (Shift Left)

Returns the result of a bitwise shift left of an operand.

Operator Shr (Shift Right)

Returns the result of a bitwise shift right of an operand.

Relational Operators



Operators that compare relationships

The relational operators perform comparisons between the values of two operands. Each operator returns a *boolean* result that is *true* (-1) if the relationship holds true, or *false* (0) if not.

Operator = (Equal)

Compares the equal relation of two operands.

Operator <> (Not Equal)

Compares the inequality relation of two operands.

Operator < (Less Than)

Compares the less than relation of two operands.

Operator <= (Less Than Or Equal)

Compares the less than or equal relation of two operands.

Operator >= (Greater Than Or Equal)

Compares the greater than or equal relation of two operands.

Operator > (Greater Than)

Compares the greater than relation of two operands.

Operator Is (Run-Time Type Information)

Checks whether an object is of a certain type.

Logical Operators



Operators that perform bitwise logic

The logical operators perform logical operations on the values of their operands, and return the results. These operators are *bitwise* operators, in that the results are found by performing logical operations on each bit of their operands.

Operator And (Conjunction)

Returns the result of a bitwise conjunction of two operands.

Operator Eqv (Equivalence)

Returns the result of a bitwise equivalence of two operands.

Operator Imp (Implication)

Returns the result of a bitwise implication of two operands.

Operator Not (Complement)

Returns the result of a bitwise complement of an operand.

Operator Or (Inclusive Disjunction)

Returns the result of a bitwise inclusive or of two operands.

Operator Xor (Exclusive Disjunction)

Returns the result of a bitwise exclusive or of two operands.

Short Circuit Operators



Operators that perform a short circuit logical evaluation.

The short circuit operators perform a evaluation on the left hand operand and depending on the result, may go on to evaluate the right hand side. The evaluations take place logically, in a comparison to zero.

Operator Andalso (Short Circuit Conjunction)

Returns the result of a short circuit conjunction of two operands.

Operator Orelse (Short Circuit Inclusive Disjunction)

Returns the result of a short circuit inclusive or of two operands.

Indexing Operators



Operators that return references based on an index

The indexing operators return references to some memory based on the value of their second, or *right-hand side*, operand. This operand is used as an index, or offset, from the beginning of some memory represented by the first, or *left-hand side*, operand.

Operator `()` (Array Index)

Returns a reference to an element in an array.

Operator `[]` (String Index)

Returns a reference to a character in a string.

Operator `[]` (Pointer Index)

Returns a reference to memory offset from a base address.

String Operators



Operators that work with strings

These operators provide conversion to string, concatenation and retrieval of character data.

Operator + (String Concatenation)

Concatenates two strings.

Operator & (String Concatenation With Conversion)

Concatenates two values converted to strings.

Operator Strptr (String Pointer)

Returns the address of a string's character data.

Preprocessor Operators



Operators that are executed by the preprocessor

These operators control how text is interpreted by the preprocessor.

Operator # (Stringize)

Returns a text operand converted to a **String** literal.

Operator ## (Concatenation)

Concatenates two text operands.

Operator ! (Escaped String Literal)

Indicates string literal immediately following must be processed for escape sequences.

Operator \$ (Non-Escaped String Literal)

Indicates string literal immediately following must not be processed for escape sequences.

Pointer Operators



Operators that work with pointers

The pointer operators provide the ability to retrieve the addresses in memory of their operands, and to use, or *dereference*, that memory.

Operator Varptr (Variable Pointer)

Returns the memory address of a variable.

Operator Strptr (String Pointer)

Returns the memory address of a string's character data.

Operator Procptr (Procedure Pointer)

Returns the memory address of a procedure.

Operator @ (Address Of)

Returns the memory address of a variable, object or procedure.

Operator * (Value Of)

Returns a reference to a variable or object at some memory address.

Type or Class Operators



Operators that work with objects

These operators return references to members of objects, given an object or its memory address.

Operator . (Member Access)

Returns a reference to a member.

Operator -> (Pointer To Member Access)

Returns a reference to a member from a pointer.

Operator Is (Run-Time Type Information)

Checks whether an object is compatible to a type derived from its runtime-type.

Memory Operators



Operators that work with memory

The memory operators provide a way to dynamically allocate and deallocate variables and objects.

Operator New

Allocates memory for and constructs objects.

Operator Placement New

Constructs objects at a specified memory location.

Operator Delete

Destroys and deallocates memory for objects.

Iterating Operators



Operators that work with iterator objects

These operators allow objects of user-defined types to be used as iterators and step values in **For . . . Next** statements.

Operator For

Allows an iterator a chance to prepare for the loop.

Operator Step

Increments an iterator object.

Operator Next

Determines if the loop should terminate or continue iterating.

Operator Step (Iteration)



Increments the iterator of a `For...Next` loop

Syntax

```
{ Type | Class | Union } typename  
Declare Operator Step (  
Declare Operator Step ( [ ByRef | ByVal ] stp As typename )  
...  
End { Type | Class | Union }
```

Usage

```
For iterator [ As typename ] = start_value To end_value [ Step s  
[ ...statements... ]  
Next
```

Parameters

typename

name of the **Type**, **Class**, or **Union**

stp, *step_value*

a *typename* object used as an incremental value

iterator

a *typename* object used as an iterator

end_value

a *typename* object used as a loop-terminating value

start_value

a *typename* object used to copy construct or assign to the iterator initial

Description

Operator For, **Operator Next** and **Operator Step** can be overloaded in and step values in `For...Next` loops.

operator step is called to increment the iterator immediately after all s

The first version of **operator step** is used if no step value is given in the loop and is passed the step value.

Example

```
' ' Example Type
Type T
  ' ' value is set by the constructor
  value As Double
  Declare Constructor( ByVal x As Double = 0 )

  Declare Operator For( ByRef stp As T )
  Declare Operator Step( ByRef stp As T )
  Declare Operator Next( ByRef cond As T, ByRef st
End Type

Constructor T ( ByVal x As Double )
  Print "T iterator constructed with value " & x
  value = x
End Constructor

Operator T.for( ByRef stp As T )
End Operator

Operator T.step( ByRef stp As T )
  Print " incremented by " & stp.value & " in step
  value += stp.value
End Operator

Operator T.next( ByRef cond As T, ByRef stp As T )
  ' ' iterator's moving from a high value to a low
  If( stp.value < 0 ) Then
    Return( value >= cond.value )
  Else
    ' ' iterator's moving from a low value to a high
    Return( value <= cond.value )
  End If
End Operator

' ' Example Usage. It looks like we are working wit
' ' have overloaded constructors. The 10, 1, and -1
```

```

For i As T = 10 To 1 Step -1
    Print i.value;
Next i

```

A more practical example demonstrating file iteration based on [cha0s](#)

```

' ' a class which iterates through files
Type FileIter
    As String pathName, fileName
    Declare Constructor( ByRef pathName As String

    Declare Operator For()
    Declare Operator Step()
    Declare Operator Next( ByRef endCond As FileIt
End Type

Constructor FileIter( ByRef pathName As String )
    this.pathName = pathName
End Constructor

Operator FileIter.for( )
    fileName = Dir(pathName & "/*.*")
End Operator

Operator FileIter.step( )
    fileName = Dir("")
End Operator

Operator FileIter.next( ByRef endCond As FileIter
    Return(fileName <> endCond.pathName)
    ' ' the c'tor sets the path name and so we chec
End Operator

' ' example code
' ' change it to any directory
For i As FileIter = "./" To ""
    Print i.fileName
Next

```

Another example working with strings:

```
Type CharIterator
  ' used to build a step var
  Declare Constructor( ByVal r As ZString Ptr )

  ' implicit step versions
  Declare Operator For ( )
  Declare Operator Step( )
  Declare Operator Next( ByRef end_cond As CharI

  ' explicit step versions
  Declare Operator For ( ByRef step_var As CharI
  Declare Operator Step( ByRef step_var As CharI
  Declare Operator Next( ByRef end_cond As CharI

  ' give the current "value"
  Declare Operator Cast( ) As String

Private:
  ' data
  value As String

  ' This member isn't necessary - we could
  ' the step variable on each iteration -
  ' but we choose this method, since we hav
  ' to compare strings otherwise. See below
  is_up As Integer
End Type

Constructor CharIterator( ByVal r As ZString Ptr )
  value = *r
End Constructor

Operator CharIterator.cast( ) As String
  Operator = value
End Operator
```

```

'' implicit step versions
''
'' In this example, we interpret implicit step
'' to always mean 'up'
Operator CharIterator.for( )
    Print "implicit step"
End Operator

Operator CharIterator.step( )
    value[0] += 1
End Operator

Operator CharIterator.next( ByRef end_cond As Char
    Return this.value <= end_cond.value
End Operator

'' explicit step versions
''
'' In this example, we calculate the direction
'' at FOR, but since the step var is passed to
'' each operator, we have the choice to also calcu
'' it "on-the-fly". For strings such as this, repe
'' may penalize, but if you're working with simple
'' then you may prefer to avoid the overhead of
'' an 'is_up' variable.
Operator CharIterator.for( ByRef step_var As CharI
    Print "explicit step"
    is_up = (step_var.value = "up")
End Operator

Operator CharIterator.step( ByRef step_var As Char
    If( is_up ) Then
        value[0] += 1
    Else
        value[0] -= 1
    End If
End Operator

Operator CharIterator.next( ByRef end_cond As Char

```

```

    If( this.is_up ) Then
        Return this.value <= end_cond.value
    Else
        Return this.value >= end_cond.value
    End If
End Operator

For i As CharIterator = "a" To "z"
    Print i; " ";
Next
Print "done"

For i As CharIterator = "a" To "z" Step "up"
    Print i; " ";
Next
Print "done"

For i As CharIterator = "z" To "a" Step "down"
    Print i; " ";
Next
Print "done"

For i As CharIterator = "z" To "a" Step "up"
    Print i; " ";
Next
Print "done"

```

Iterating with fractions:

```

Type fraction
    ' Used to build a step var
    Declare Constructor( ByVal n As Integer, ByVal

    ' Implicit step versions
    Declare Operator For ( )
    Declare Operator Step( )
    Declare Operator Next( ByRef end_cond As fract

```

```

'' Explicit step versions
Declare Operator For ( ByRef step_var As fract
Declare Operator Step( ByRef step_var As fract
Declare Operator Next( ByRef end_cond As fract

'' Give the current "value"
Declare Operator Cast( ) As Double
Declare Operator Cast( ) As String

Private:
    As Integer num, den
End Type

Constructor fraction( ByVal n As Integer, ByVal d
    this.num = n : this.den = d
End Constructor

Operator fraction.cast( ) As Double
    Operator = num / den
End Operator

Operator fraction.cast( ) As String
    Operator = num & "/" & den
End Operator

'' Some fraction functions
Function gcd( ByVal n As Integer, ByVal m As Integ
    Dim As Integer t
    While m <> 0
        t = m
        m = n Mod m
        n = t
    Wend
    Return n
End Function

Function lcd( ByVal n As Integer, ByVal m As Integ
    Return (n * m) / gcd( n, m )
End Function

```

```

''
'' Implicit step versions
''
'' In this example, we interpret implicit step
'' to mean 1
''
Operator fraction.for( )
    Print "implicit step"
End Operator

Operator fraction.step( )
    Var lowest = lcd( this.den, 1 )

    Var mult_factor = this.den / lowest
    Dim As fraction step_temp = fraction( 1, 1 )

    this.num *= mult_factor
    this.den *= mult_factor

    step_temp.num *= lowest
    step_temp.den *= lowest

    this.num += step_temp.num
End Operator

Operator fraction.next( ByRef end_cond As fraction
    Return This <= end_cond
End Operator

''
'' Explicit step versions
''
Operator fraction.for( ByRef step_var As fraction
    Print "explicit step"
End Operator

Operator fraction.step( ByRef step_var As fraction
    Var lowest = lcd( this.den, step_var.den )

```

```

Var mult_factor = this.den / lowest
Dim As fraction step_temp = step_var

this.num *= mult_factor
this.den *= mult_factor

mult_factor = step_temp.den / lowest

step_temp.num *= mult_factor
step_temp.den *= mult_factor

this.num += step_temp.num
End Operator

Operator fraction.next( ByRef end_cond As fraction
    If(( step_var.num < 0 ) Or ( step_var.den < 0
        Return This >= end_cond
    Else
        Return This <= end_cond
    End If
End Operator

For i As fraction = fraction(1,1) To fraction(4,1)
    Print i; " ";
Next
Print "done"

For i As fraction = fraction(1,4) To fraction(1,1)
    Print i; " ";
Next
Print "done"

For i As fraction = fraction(4,4) To fraction(1,4)
    Print i; " ";
Next
Print "done"

For i As fraction = fraction(4,4) To fraction(1,4)
    Print i; " ";

```

```
Next  
Print "done"
```

Dialect Differences

- Only available in the *-lang fb* dialect.

See also

- [Operator For](#)
- [Operator Next](#)
- [For...Next](#)

Operator Next (Iteration)



Determines if a **For...Next** loop should be terminated

Syntax

```
{ Type | Class | Union } typename  
Declare Operator Next ( [ ByRef | ByVal ] cond As typename ) As  
Integer  
Declare Operator Next ( [ ByRef | ByVal ] cond As typename, [ ByRef | ByVal ] stp As typename ) As Integer  
...  
End { Type | Class | Union }
```

Usage

```
For iterator [ As typename ] = start_value To end_value [ Step step_value ]  
[ ...statements... ]  
Next
```

Parameters

typename

name of the **Type**, **Class**, or **Union**

cond, *end_value*

a *typename* object used as a loop-terminating value

stp, *step_value*

a *typename* object used as an incremental value

iterator

a *typename* object used as an iterator

start_value

a *typename* object used to copy construct or assign to the iterator initially

Description

Operator For, **Operator Next** and **Operator Step** can be overloaded in user-defined type definitions to allow objects of that type to be used as iterators and step values in **For...Next** loops.

operator Next is called every time the iterator needs to be checked

against the end value. This happens immediately after the call to its **Operator For**, and immediately after any calls to its **Operator Step**. **Operator Next** should return zero (0) if the loop should be terminated, or non-zero if the loop should continue iterating. The first time **Operator Next** is called, no statements in the **For . . . Next** body, if any, have been executed yet.

The first version of **Operator Next** is used if no step value is given in the **For . . . Next** statement. If a step value is given, the second version is used and is passed the step value.

Example

See the **Operator Step** examples.

Dialect Differences

- Only available in the *-lang fb* dialect.

See also

- **Operator For**
- **Operator Step**
- **For . . . Next**

Operator Precedence



When several operations occur in a single expression, each operation is evaluated and resolved in a predetermined order. This is called the order of operation or operator precedence.

If an operator in an expression has a higher precedence, it is evaluated before an operator of lower precedence.

If operators have equal precedence, they then are evaluated in the order in of their associativity. The associativity may be Left-to-Right or Right-to-Left order.

As a rule, binary operators (such as `+`, `^`) and unary postfix operators (such as `()`, `->`) are evaluated Left-to-Right, and unary prefix operators (such as `Not`, `@`) are evaluated Right-to-Left.

Operators that have an associativity of "N/A" indicate that there is no expression in which the operator can be used where its order of operation would need to be checked, either by precedence or by associativity. Function-like operators such as `cast` are always the first to be evaluated due to the parentheses required in their syntax. And assignment operators are always the last to be evaluated.

Parentheses can be used to override operator precedence. Operations within parentheses are performed before other operations. Within the parentheses normal operator precedence is used.

The following table lists operator precedence from highest to lowest. Breaks in the table mark the groups of operators having equal precedence.

Highest Precedence

Operator	Description	Associativity
CAST	Type Conversion	N/A

PROCPTR	Procedure pointer	N/A
STRPTR	String pointer	N/A
VARPTR	Variable pointer	N/A
[]	String index	Left-to-Right
[]	Pointer index	Left-to-Right
()	Array index	Left-to-Right
()	Function Call	Left-to-Right
.	Member access	Left-to-Right
->	Pointer to member access	Left-to-Right
@	Address of	Right-to-Left
*	Value of	Right-to-Left
New	Allocate Memory	Right-to-Left
Delete	Deallocate Memory	Right-to-Left
^	Exponentiate	Left-to-Right
-	Negate	Right-to-Left
*	Multiply	Left-to-Right
/	Divide	Left-to-Right
\	Integer divide	Left-to-Right
MOD	Modulus	Left-to-Right
SHL	Shift left	Left-to-Right
SHR	Shift right	Left-to-Right
+	Add	Left-to-Right
-	Subtract	Left-to-Right
&	String concatenation	Left-to-Right
Is	Run-time type information check	N/A

=	Equal	Left-to-Right
<>	Not equal	Left-to-Right
<	Less than	Left-to-Right
<=	Less than or equal	Left-to-Right
>=	Greater than or equal	Left-to-Right
>	Greater than	Left-to-Right
NOT	Complement	Right-to-Left
AND	Conjunction	Left-to-Right
OR	Inclusive Disjunction	Left-to-Right
EQV	Equivalence	Left-to-Right
IMP	Implication	Left-to-Right
XOR	Exclusive Disjunction	Left-to-Right
ANDALSO	Short Circuit Conjunction	Left-to-Right
ORELSE	Short Circuit Inclusive Disjunction	Left-to-Right
=[>]	Assignment	N/A
&=	Concatenate and Assign	N/A
+=	Add and Assign	N/A
-=	Subtract and Assign	N/A
*=	Multiply and Assign	N/A
/=	Divide and Assign	N/A
\=	Integer Divide and Assign	N/A
^=	Exponentiate and Assign	N/A
MOD=	Modulus and Assign	N/A
AND=	Conjunction and Assign	N/A
EQV=	Equivalence and Assign	N/A
IMP=	Implication and Assign	N/A
OR=	Inclusive Disjunction and Assign	N/A
XOR=	Exclusive Disjunction and Assign	N/A
SHL=	Shift Left and Assign	N/A

SHR=	Shift Right and Assign	N/A
LET	Assignment	N/A
LET()	Assignment	N/A

In some cases, the order of precedence can cause confusing or counter intuitive results. Here are some examples:

```
' trying to raise a negated number to a power
```

```
-2 ^ 2
```

```
Desired result: (-2) ^ 2 = 4
```

```
Actual result: -(2 ^ 2) = -4
```

```
' trying to test a bit in a number
```

```
n And 1 <> 0
```

```
Desired result: (n And 1) <> 0
```

```
Actual result: n And (1 <> 0)
```

```
' trying to shift a number by n+1 bits
```

```
a Shl n+1
```

```
Desired result: a Shl (n + 1)
```

```
Actual result: (a Shl n) + 1
```

For expressions where the operator precedence may be ambiguous, it is recommended to wrap parts of the expression in parentheses, in order both to minimise the possibility of error and to aid comprehension for people reading the code.

See also

- [Operators](#)

Bitwise Operators Truth Tables



Computed values for the bitwise logical operators.

Binary operators

Operators that take two operands.

Unary operator

Operator that take a single operand.

These logical operators return a value based on the value of their operand(s). For the binary operators, each bit in the left-hand side value is applied logically to the corresponding bit in the right-hand side value. The result of this operation is returned. For the unary operator, (**Operator Not**), the logic is applied to its right-hand side operand only.

Binary operators

Operator And (Conjunction)

Bits in the result are set if and only if both of the corresponding bits in the left and right-hand side operands are set.

Lhs	0	0	1	1
Rhs	0	1	0	1
Result	0	0	0	1

Operator Eqv (Equivalence)

Bits in the result are set if and only if both of the corresponding bits in the left and right-hand side operands are both either set or unset.

--	--	--	--	--

Operator Xor (Exclusive Disjunction)

Bits in the result are set if and only if one of the corresponding bits in the left and right-hand side operands is set.

Lhs	0	0	1	1
Rhs	0	1	0	1
Result	0	1	1	0

Unary operators

Operator Not (Complement)

Bits in the result are set if the corresponding bits in the right-hand side operand are unset, and unset if they are set.

--	--	--	--

Lhs	0	0	1	1
Rhs	0	1	0	1
Result	1	0	0	1

Rhs	0	1
Result	1	0

Operator Imp (Implication)

Bits in the result are set if and only if the corresponding bit in the left-hand side operand implies the bit in the right-hand side operand.

Lhs	0	0	1	1
Rhs	0	1	0	1
Result	1	1	0	1

Operator Or (Inclusive Disjunction)

Bits in the result are set if either of the corresponding bits in the left and right-hand side operands are set.

Lhs	0	0	1	1
Rhs	0	1	0	1
Result	0	1	1	1

Control Flow Statements



Statements that direct the flow of program execution.

Transferring Statements

Statements that transfer control to another part of a program.

Branching Statements

Statements that execute one of a number of code branches.

Looping Statements

Statements that execute code repeatedly.

Transferring Statements

Goto

Transfers execution to another point in code defined by a text label.

GoSub

Temporarily transfers execution to another point in code, defined by a text label.

On Goto

Transfers execution to one of a number of points in code defined by text labels, based on the value of an expression.

On Gosub

Temporarily transfers execution to one of a number of points in code defined by text labels, based on the value of an expression.

Return

Returns from a call using **GoSub** or from a procedure returning a value.

Branching Statements

If..End If

Looping Statements

While..Wend

Executes a block of statements while a condition is met.

For..Next

Executes a block of statements while an iterator is less than or greater than an expression.

Do..Loop

Executes a block of statements while or until a condition is met.

Intra-loop control

Continue While, Continue For and Continue Do

Prematurely re-enters a loop.

Exit While, Exit For and Exit Do

Prematurely breaks out of a loop.

Executes a block of statements if a condition is met.

..Else If..

Executes a block of code if a condition is met and all previous conditions weren't met.

..Else..

Executes a block of code if all previous conditions weren't met.

Select..End Select

Executes one of a number of statement blocks using a set of conditions.

..Case..

Executes a block of code if a condition is met.

..Case Else..

Executes a block of code if all previous conditions weren't met.

Intra-branch control

Exit Select

Prematurely breaks out of a **Select..End Select** statement.

Keywords that work with procedures.

Description

These keywords control the declaration and definition of both module-level procedures and member procedures, how they are called, how arguments are passed and how their names are seen externally to other modules. Procedures can also be declared to be executed automatically before any module-level code is executed.

Declaration

Keywords that declare and define procedures.

Linkage

Keywords that specify how procedure names are seen by external modules.

Calling conventions

Keywords that specify how arguments are used when calling procedures.

Parameter passing conventions

Keywords that specify how arguments are passed to procedures.

Variadic Procedures

Macros that allow for an arbitrary number of arguments to be passed a procedure.

Automatic execution

Keywords that specify automatic execution of procedures.

Miscellaneous

Miscellaneous keywords.

Declaration

Declare

Declares a module-level or member procedure.

Sub

Specifies a procedure that does not return an argument.

Parameter passing conventions

ByRef

Specifies passing an argument by reference.

ByVal

Specifies passing an argument

Function

Specifies a procedure that returns an argument.

Overload

Specifies that the procedure name can be used in other procedure declarations.

Static

Specifies static storage for all variables and objects in the procedure body.

Const (Member)

Specifies a const member procedure in user-defined type definitions.

Static (Member)

Specifies a static member procedure in user-defined type definitions.

Linkage

Public

Specifies external linkage for a procedure.

Private

Specifies internal linkage for a procedure.

Alias

Specifies an alternate external name for a procedure.

Export

Specifies a procedure is to be exported from a shared library.

Lib

Specifies automatic loading of a library.

Calling conventions

stdcall

by value.

Any

Disables type-checking on arguments.

Variadic Procedures

... (Ellipsis)

Indicates a variadic procedure in a declaration.

va_first

Macro to obtain the argument list in a variadic procedure.

va_arg

Macro to obtain the current argument in a variadic procedure.

va_next

Macro to move to the next argument in a variadic procedure.

Automatic execution

Constructor (Module)

Indicates a procedure is to be executed before module-level code.

Destructor (Module)

Indicates a procedure is to be executed after module-level code.

Miscellaneous

Byref (Function Results)

Specifies that a function returns by reference rather than by value.

Call

Invokes a procedure.

Naked

Specifies the standard calling convention for BASIC languages, including FreeBASIC.

cdecl

Specifies the standard calling convention in the C and C++ languages.

pascal

Specifies the standard calling convention in the Fortran, Pascal and Microsoft QuickBASIC/QBasic languages.

Specifies that a function body is not to be given any prolog/epilog code

Modularizing



Keywords helpful when writing modular programs.

- **Common**
- **DyLibFree**
- **DyLibLoad**
- **DyLibSymbol**
- **Export**
- **Extern**
- **Extern...End Extern**
- **Import**
- **Namespace**
- **Private**
- **Public**
- **Using (Namespaces)**

Commands that control the preprocessor.

Description

Preprocessor commands are sent to the compiler to control what gets compiled and how. They can be used to choose to compile one block of code rather than another for cross-platform compatibility, include headers or other source files, define small inline functions called macros, or alter how the compiler handles variables.

Conditional Compilation

Commands that allow for branches in compilation based on conditions:

Text Replacement

Commands that create text-replacement macros.

File Directives

Commands that indicate to the compiler how other files relate to the source file.

Control Directives

Commands that set compile options, control compilation, and report compile time information.

Metacommands

Commands that are kept for backward compatibility.

Conditional Compilation

#if

Compiles the following code block based on a condition.

#ifdef

Compiles the following code block if a symbol is defined.

#ifndef

Compiles the following code block if a symbol is not defined.

#elseif

Compiles the following code

File Directives

#include

Inserts text from a file.

#inlib

Includes a library in the linking processes.

#libpath

Includes a path to search for libraries in the linking process.

Control Directives

#pragma

block if a condition is true and the previous conditions was false.

#else

Compiles the following code block if previous conditions were false.

#endif

Signifies the end of a code block.

defined

Returns "-1" if a symbol is defined, otherwise "0".

Text Replacement

#define

Creates a single-line text-replacement macro.

#macro and #endmacro

Creates a multi-line text-replacement macro.

#undef

Undefines a symbol.

Preprocessor Stringize

Converts text into a string literal.

Preprocessor Concatenate

Concatenates two pieces of text.

! Escaped String Literal

Indicates string literal immediately following must be processed for escape sequences.

\$ Non-Escaped String Literal

Indicates string literal immediately following must not be processed for escape sequences.

Sets compiling options.

#lang

Sets dialect from source.

#print

Outputs a messages to standard output while compiling.

#error

Outputs a messages to standard output and stops compilation.

#Assert

Stops compilation with an error message if a given condition is false.

#line

Sets the current line number and file name.

Metacommands

'\$Include

Alternate form of the **#include** directive.

'\$Dynamic

Alternate form of the **Option Dynamic** statement.

'\$Static

Alternate form of the **Option Static** statement.

'\$Lang

Alternate form of the **#lang** directive.

Escape Sequences



Escape sequences can be used in string literals by using the operator !

Usage

```
result = !"text"
```

Description

The accepted escape sequences in *text* are:

\a	beep
\b	backspace
\f	formfeed
\l or \n	newline
\r	carriage return
\t	tab
\unnnn	unicode char in hex
\v	vertical tab
\nnn	ascii char in decimal
\&hnn	ascii char in hex
\&onnn	ascii char in octal
\&bnnnnnnnn	ascii char in binary
\\	backslash
\(double quote)	double quote
\'	single quote

Note: The zero-character (\000 = \&h00; = \&o000; = \&b00000000;) is the null terminator. Only characters before the first null terminator can be seen when the literal is used as a **string**. To get a zero character in a string use **chr(0)** instead.

See also

- **Operator ! (Escaped String)**

- **Operator \$ (Non-Escaped String)**
- **Option Escape**
- **String**
- **Chr**
- **Literals**

Compiler Switches



Statements that affect how code is compiled.

Description

These statements affect how the compiler declares variables, arrays and procedures, parses string literals, passes procedure parameters and more.

Metacommands

- `'$Dynamic`
- `'$Include`
- `'$Static`
- `'$Lang`

Compiler Options

- `Option Base`
- `Option ByVal`
- `Option Dynamic`
- `Option Escape`
- `Option Explicit`
- `Option GOSUB`
- `Option NOGOSUB`
- `Option NOKEYWORD`
- `Option Private`
- `Option Static`

Set Default Datatypes

- `DefByte`
- `DefDbL`
- `DefInt`
- `DefLng`
- `Deflongint`
- `DefShort`
- `DefSng`
- `DefStr`
- `DefUByte`
- `DefUInt`
- `Defulongint`
- `DefUShort`

Dialect Differences

- `Deflongint` and `Defulongint` available only in the *-lang fblite* dialect.

- OPTION statements are available only in the *-lang fblite* and *-lang qb* dialects only.

See also

- **Preprocessor**

Preprocessor symbols defined by the compiler.

Description

Intrinsic defines are set by the compiler and may be used as any other defined symbol. Intrinsic defines often convey information about the state of the compiler, either in general or at a specific point in the compilation process. Most intrinsic defines are associated with a value.

Platform Information

Defines that provide information on the system.

Version Information

Defines that provide information on the `fbcc` compiler version being used.

Command-line switches

Defines that provide information with the command-line switches used with `fbcc`.

Environment Information

Defines that provide information about the operating system environment.

Context-specific Information

Defines that provide context information about the compilation process.

Platform Information

__FB_WIN32__

Defined if compiling for Windows.

__FB_LINUX__

Defined if compiling for Linux.

__FB_DOS__

Defined if compiling for DOS.

__FB_CYGWIN__

Defined if compiling for Cygwin.

__FB_FREEBSD__

Defined if compiling for

Environment Information

__FB_ARGC__

Defined as an integer literal of the number of command-line arguments passed to the program.

__FB_ARGV__

Defined as a `ZString Ptr Ptr` to the command line arguments passed to the program.

__DATE__

Defined as a string literal of the

FreeBSD.

__FB_NETBSD__

Defined if compiling for NetBSD.

__FB_OPENBSD__

Defined if compiling for OpenBSD.

__FB_DARWIN__

Defined if compiling for Darwin.

__FB_XBOX__

Defined if compiling for Xbox.

__FB_BIGENDIAN__

Defined if compiling on a system using big-endian byte-order.

__Fb_Pcos__

Defined if compiling for a common PC OS (e.g. DOS, Windows, OS/2).

__Fb_Unix__

Defined if compiling for a Unix-like OS.

__Fb_64Bit__

Defined if compiling for a 64bit target.

__Fb_Arm__

Defined if compiling for the ARM architecture.

Version Information

__FB_VERSION__

Defined as a string literal of the compiler version.

__FB_VER_MAJOR__

Defined as an integral literal of the compiler major version number.

__FB_VER_MINOR__

Defined as an integral literal of the compiler minor version

compilation date in "mm-dd-yyyy" format.

__Date_Iso__

Defined as a string literal of the compilation date in "yyyy-mm-dd" format.

__TIME__

Defined as a string literal of the compilation time.

__PATH__

Defined as a string literal of the absolute path of the module.

Context-specific Information

__FILE__ and **__FILE_NQ__**

Defined as the name of the module.

__FUNCTION__ and

__FUNCTION_NQ__

Defined as the name of the procedure where it's used.

__LINE__

Defined as an integer literal of the line of the module where it's used.

__FB_OPTION_BYVAL__

True (-1) if parameters are declared by value by default, zero (0) otherwise.

__FB_OPTION_DYNAMIC__

True (-1) if all arrays are variable-length, zero (0) otherwise.

__FB_OPTION_ESCAPE__

True (-1) if string literals are processed for escape sequences, zero (0) otherwise.

__Fb_Option_Gosub__

True (-1) if gosub support is

number.

__FB_VER_PATCH__

Defined as an integral literal of the compiler patch number.

__FB_MIN_VERSION__

Macro to check for a minimum compiler version.

__FB_BUILD_DATE__

Defined as a string literal of the compiler build date.

__FB_SIGNATURE__

Defined as a string literal of the compiler signature.

Command-line switches

__Fb_Asm__

Defined to either "intel" or "att" depending on **-asm**.

__Fb_Backend__

Defined to either "gas" or "gcc" depending on **-gen**.

__Fb_Gcc__

True (-1) if -gen gcc is used, false (0) otherwise.

__FB_MAIN__

Defined if compiling a module with an entry point.

__FB_DEBUG__

True (-1) if the "-g" switch was used, false (0) otherwise.

__FB_ERR__

Zero (0) if neither the "-e", "-ex" or "-exx" switches were used.

__Fb_Fpmode__

Defined as "fast" if compiling for fast SSE math, "precise" otherwise.

__Fb_Fpu__

enabled, zero (0) otherwise.

__FB_OPTION_EXPLICIT__

True (-1) if variables and objects need to be explicitly declared, zero (0) otherwise.

__FB_OPTION_PRIVATE__

True (-1) if all procedures are private by default, zero (0) otherwise.

Defined as "sse" if compiling for SSE floating point unit, or "x87" for normal x87 floating-point unit.

__FB_LANG__

Defined to a string literal of the "-lang" dialect used.

__FB_MT__

True (-1) if the "-mt" switch was used, false (0) otherwise.

__FB_OUT_DLL__

True (-1) in a module being compiled and linked into a shared library, false (0) otherwise.

__FB_OUT_EXE__

True (-1) in a module being compiled and linked into an executable, false (0) otherwise.

__FB_OUT_LIB__

True (-1) in a module being compiled and linked into a static library, zero (0) otherwise.

__FB_OUT_OBJ__

True (-1) in a module being compiled only, zero (0) otherwise.

__FB_SSE__

Defined if compiling for SSE floating point unit.

__Fb_Vectorize__

Defined as the level of automatic vectorization (0 to 2)

Handling runtime errors.

FreeBASIC can handle the errors in the following ways:

- By default the program does nothing with the errors - they are silent and code continues. In this case code should process possible errors by using the `Err` function.
- If compiled with `-e` or `-ex` options, FreeBASIC uses QB-like error handling.
- **Future** OOP versions of FreeBASIC may have a java-like TRY..CATCH exception handler implemented.

NOTE: The following information is valid unless the error produces an Access Protection Fault (for example if the program writes outside the process memory). In these cases the OS will immediately stop the program and issue an error message. Avoid it from inside FreeBASIC.

Default error handling

The default FreeBASIC behavior is to set the ERR variable and continue.

```
Dim As Integer e
Open "xzxwz.zwz" For Input As #1
e = Err
Print e
Sleep
```

(The example program supposes there is no `xzxwz.zwz` file). The program will stop; it sets the ERR variable and continues. The error can be processed on the next line.

Some IO functions such as `Open` and `Put #...` can be used in function form to return an error number or zero if successful.

```
Print Open ("xzxwz.zwz" For Input As #1)
Sleep
```

QuickBASIC-like error handling

If the **-e** or **-ex** switch is used at compile time, the program is expected like error handler enabled. If no handler processes the error, the program error.

Notice: if QB-Like error handling is used, the programmer should be prepared to handle all error conditions.

```
' ' Compile with QB (-lang qb) dialect
'$lang: "qb"

On Error Goto FAILED
Open "xzxwz.zwz" For Input As #1
On Error Goto 0
Sleep
End

FAILED:
Dim e As Integer
e = Err
Print e
Sleep
End
```

On Error sets an error handling routine which the program will jump to if found. **On Error Goto 0** disables the error handling.

If an error handling routine is not set when an error occurs, the program will send the console an error message.

```
Aborting program due to runtime error 2 (file not found)
```

The error handler routine can be at the end of the program, as in QB. **Error** statement allows the setting of a local error handler routine at the same **Sub** or **Function** in which the error occurs.

```
' ' Compile with -e
' ' The -
e command line option is needed to enable error ha

Declare Sub foo
  foo
  Sleep

Sub foo

  Dim filename As String
  Dim errmsg As String
  filename = ""
  On Local Error Goto fail
  Open filename For Input Access Read As #1
  Print "No error"
  On Local Error Goto 0
  Exit Sub

fail:
errmsg = "Error " & Err & _
        " in function " & *Erfn & _
        " on line " & Erl
Print errmsg

End Sub
```

If the **-e** switch is used (whatever the **-lang** dialect), the error handler

the program.

With **-ex** and **-lang qb** dialect only, the error routine can end by using the statement that caused the error) or **Resume Next** (continues at the |

Error codes

See **Runtime Error Codes** for a listing of runtime error numbers and meaning.

No user error code range is defined. If **Error** is used to set an error code use high values to avoid collisions with the list of built-in error codes. (may be expanded later.)

See also

- **Error Handling Functions**
- **Runtime Error Codes**

Array Functions



Statements and procedures for working with arrays.

Defining Arrays

Statements that create arrays.

Clearing Array Data

Procedures that work with array memory.

Retrieving Array Size

Procedures that return bounds of an array's dimension.

Defining Arrays

Option Dynamic

Forces arrays to be defined as variable-length arrays.

'\$Dynamic

Alternate form of the **Option Dynamic** statement.

Option Static

Reverts a previous **Option Dynamic** command.

'\$Static

Alternate form of the **Option Static** statement.

ReDim

Defines and resizes variable-length arrays.

Preserve

Preserves array contents when used with **ReDim**.

Clearing Array Data

Erase

Destroys variable-length array elements and initializes fixed-length array elements.

Retrieving Array Size

LBound

Returns the lower bound of an array's dimension.

UBound

Returns the upper bound of an array's dimension.

Macros that work with the bits and bytes of numbers.

Description

The macros documented here provide access to the individual bits, bytes and words of integer values.

Byte Manipulation Macros

Gets the value of individual bytes or words of **uInteger** values.

Bit Manipulation Macros

Gets the state of individual bits of numeric values.

Byte Manipulation Macros

LoByte

Gets the least significant byte (LSB, or *lo-byte*) value of an **uInteger** value.

HiByte

Gets the most significant byte (MSB, or *hi-byte*) value of the least significant word (LSW, or *lo-word*) of an **uInteger** value.

LoWord

Gets the least significant word (LSW, or *lo-word*) value of an **uInteger** value.

HiWord

Gets the most significant word (MSW, or *hi-word*) value of an **uInteger** value.

Bit Manipulation Macros

Bit

Gets the state of an individual bit in an integer value.

BitReset

Gets the value of an integer with a specified bit cleared.

BitSet

Gets the value of an integer with a specified bit set.

Procedures that work with the console.

Description

These procedures provide ways to output text to the console, as well as control where and how text is output.

Configuring the Console

Statements that affect how text is displayed.

Cursor Color and Positioning

Procedures that move the cursor and change its color.

Writing Text to the Console

Procedures that output text to the console.

Configuring the Console

Cls

Clears the entire screen or text viewport.

Width

Sets or returns the number of rows and columns of the console display.

View Print

Sets the printable area of the console screen.

Cursor Color and Positioning

Color

Changes the foreground and background color of text to be written.

CsrLin

Returns the row position of the cursor.

Pos

Writing Text to the Console

Print

?

Writes text to the console.

Print Using

? Using

Writes formatted text to the console.

Write

Writes a list of items to the console.

Spc

Skips a number of spaces when writing text.

Tab

Skips to a certain column when writing text.

Returns the column position of the cursor.

Locate

Sets the row and column position of the cursor and its visibility.

Screen (Console)

Gets the character or color attribute at a given location.

Date and Time Functions



Procedures that work with dates and time.

Description

These procedures provide ways to deal with date and time intervals in a consistent way. Additional procedures are provided to set and get the current system date and time, and to retrieve a time stamp for sensitive timing algorithms.

VisualBasic compatible procedures

Procedures for working with so-called **date serials**, similar to those used in Visual Basic(r).

Date and time procedures

Procedures for working with the system date and time.

VisualBasic compatible procedures

Now

Gets a **date serial** of the current date and time.

Creating Date serials

DateSerial

Gets the **date serial** representation of a date.

TimeSerial

Gets the **date serial** representation of a time.

DateValue

Gets the **date serial** representation of a date expressed as a **string**.

TimeValue

Gets the **date serial** representation of a time

Date and time procedures

Date

Gets the **string** representation of the current system date.

Time

Gets the **string** representation of the current system time.

SetDate

Sets the current system date.

SetTime

Sets the current system time.

Timer

Gets a counter expressed in seconds.

expressed as a **String**.

Extracting information from Date serials

Second

Gets the seconds of the hour from a **date serial**.

Minute

Gets the minutes of the hour from a **date serial**.

Hour

Gets the hour of the day from a **date serial**.

Day

Gets the day of the month from a **date serial**.

Weekday

Gets the day of the week from a **date serial**.

Month

Gets the month of the year from a **date serial**.

Year

Gets the year from a **date serial**.

DatePart

Gets a time interval from a **date serial**.

Extracting information from Date serials

DateAdd

Gets the result of a time interval added to a **date serial**.

DateDiff

Gets a time interval between two **date serials**.

Miscellaneous

IsDate

Tests if a `String` can be converted to a `date serial`.

MonthName

Gets the month name of its integer representation.

WeekdayName

Gets the weekday name of its integer representation.

Error Handling Functions



Statements and procedures that provide runtime error-handling capabilities.

Description

These statements and procedures provide ways of dealing with runtime errors. Specific modules, procedures and source code lines can be retrieved, and error handlers can be set up.

Determining Errors

Procedures that retrieve information about an error.

Handling Errors

Statements that allow handling of errors.

Determining Errors

Erl

Gets the line in source code where the error occurred.

Erfn

Gets the name of the function where the error occurred.

Ermn

Gets the name of the source file where the error occurred.

Err

Gets the error number of the last error that occurred.

Error

Generates an error using an error number.

Handling Errors

On Error

Sets a global error handler using a label.

On Local Error

Sets a local error handler using a label.

Resume

Resumes execution at the line where the error occurred.

Resume Next

Resumes execution at the line after where the error occurred.

See also

- **Error Handling**
- **Runtime Error Codes**

Statements and procedures for working with files and devices.

Description

These statements and procedures provide file and device i/o capabilities. So called *file numbers* can be bound to files or devices, which can be read or written to using formatted (text mode) or unformatted (binary mode) data. In binary mode, files and devices can be read from or written to in arbitrary locations. For multithreaded applications, files and devices can also be locked.

Opening Files or Devices

Procedures and other keywords that provide read or write access to a file or device.

Reading from and Writing to Files or Devices

Procedures that read and write data to an opened file or device.

File Position and other Info

Procedures that determine where reading and writing will take place within an opened file.

Opening Files or Devices

FreeFile

Gets an available file number that can be used to read or write from files or devices.

Open

Binds a file number to a physical file to provide reading and writing capabilities.

Open Com

Binds a file number to a communications port.

Open Cons

Binds a file number to the standard input and output

Reading from and Writing to Files or Devices

Input #

Reads a list of values from a file or device.

Write #

Writes a list of values to a file or device.

Input()

Reads a number of characters from a file or device.

Winput()

Reads a number of wide characters from a file or device.

Line Input #

streams.

Open Err

Binds a file number to the standard input and error streams.

Open Lpt

Binds a file number to a printer device.

Open Pipe

Binds a file number to the input and output streams of a process.

Open Scrn

Binds a file number directly to the console.

Close

Unbinds a file number from a file or device.

Reset

Unbinds all active file numbers.

File I/O modes

Input (File Mode)

Text data can be read from the file.

Output

Text data can be written to the file.

Append

Text data is added to the end of a file when output.

Binary

Arbitrary data can be read from or written to the file.

Random

Blocks of data of certain size can be read from and written to the file.

Reads a line of text from a file or device.

Print #

? #

Writes text data to a file or device.

Put #

Writes arbitrary data to a file or device.

Get #

Reads arbitrary data from a file or device.

File Position and other Info

LOF

Gets the length (in bytes) of a file.

LOC

Gets the file position of the last read or write operation.

EOF

Returns true if all of the data has been read from a file.

Seek (Statement)

Sets the file position of the next read or write operation.

Seek (Function)

Gets the file position of the next read or write operation.

Lock

Restricts read or write access to a file or portion of a file.

Unlock

Remove read or write restrictions from a previous

Lock command.

File access privileges

Access

An overview of file access privileges.

Read (File Access)

Binary data can only be read from the file.

Write (File Access)

Binary data can only be written to the file.

Read Write(File Access)

Binary data can be read from and written to the file.

Character encoding

Encoding

Specifies the character encoding of a file.

Mathematical Functions



Procedures that work with numbers mathematically.

Description

This set of procedures provide basic algebraic and trigonometric function. Random numbers can also be retrieved, using a variety of random number generators.

Algebraic Procedures

Absolute values, logarithms, square roots and more.

Trigonometry Procedures

Sine, Cosine and other trigonometry-related procedures.

Miscellaneous Procedures

Miscellaneous procedures.

Algebraic Procedures

Abs

Returns the absolute value of a number.

Exp

Returns e raised to some power.

Log

Returns the natural logarithm of a number.

Sqr

Returns the square root of a number.

Fix

Returns the integer part of a number.

Frac

Returns the fractional part of a number.

Int

Returns the largest integer less

Trigonometric Procedures

Sin

Returns the sine of an angle.

Asin

Returns the arcsine of a number.

Cos

Returns the cosine of an angle.

Acos

Returns the arccosine of a number.

Tan

Returns the tangent of an angle.

Atn

Returns the arctangent of a number.

Atan2

Returns the arctangent of the ratio between two numbers.

Miscellaneous Procedures

than or equal to a number.

Sgn

Returns the sign of a number.

Randomize

Seeds the random number generator used by **Rnd**.

Rnd

Returns a random **Double** in the range [0, 1).

Procedures that work with static and dynamic memory.

Description

These procedures provide access to the free store, or heap. Memory from the free store can be reserved and freed, and procedures are provided to read and write directly to that memory.

Working with Dynamic Memory

Procedures that reserve, resize or free dynamic memory.

Miscellaneous Procedures

Procedures that read or write values to and from addresses in memory.

Working with Dynamic Memory

Allocate

Reserves a number of bytes of uninitialized memory and returns the address.

CAllocate

Reserves a number of bytes of initialized (zeroed) memory and returns the address.

Reallocate

Changes the size of reserved memory.

Deallocate

Returns reserved memory back to the system.

Miscellaneous Procedures

Peek

Reads some type of value from an address.

Poke

Writes some type of value to an address.

Clear

Clears data in an array with a specified value.

Swap

Exchange the contents of two variables.

SAdd

Returns the address for the data in a string variable.

Operating System Functions



Statements and procedures for working with files, directories and the system.

Description

The statements and procedures listed here provide access to the operating system environment. They transfer execution to external programs, get information about files and directories, manipulate the file system and send commands to the command shell.

Working with Files

Procedures that deal with files.

Working with Directories

Various directory management procedures.

File Properties

Get information about files.

System Procedures

Procedures for working with the environment.

Working with Files

Exec and Chain

Temporarily transfers control to another program.

Run

Transfers control to another program.

Kill

Deletes an existing file.

Name

Renames an existing file.

Working with Directories

CurDir

Gets the current working directory.

File Properties

FileAttr

Gets information about a file bound to a file number.

FileCopy

Copies a file.

FileDateTime

Gets the last modified date and time of a file.

FileExists

Tests for the existence of a file.

FileLen

Gets the length (in bytes) of a file.

System Procedures

ChDir

Sets the current working directory.

Dir

Gets the names of files or directories matching certain attributes.

ExePath

Gets the directory of the current running program.

MkDir

Creates a new directory.

RmDir

Deletes an existing directory.

Fre

Gets the amount of free memory (in bytes) available.

Command

Gets the command-line parameters passed to the program.

Environ

Gets the value of an environment variable.

Isredirected

Checks whether stdin or stdout is redirected to a file or not.

SetEnviron

Sets the value of an environment variable.

Shell

Sends a command to the system command interpreter.

System

Closes all open files and exits the program.

Statements and Procedures that work with strings.

Description

These statements and procedures provide many ways to create and manipulate strings and substrings. Numbers can be converted to strings and vice-versa. Procedures are also provided to aid in serialization of numeric data, perhaps for persistent storage.

Creating Strings

String data types and procedures that create new strings.

Character Conversions

Procedures that convert from character codes to strings and back.

Numeric/Boolean to String Conversions

Procedures that convert numeric values to strings.

String to Numeric Conversions

Procedures that convert strings to numeric values.

Numeric Serializations

Procedures that convert raw numeric data to and from strings suitable for storage.

Working with Substrings

Procedures that return subsets of strings, or that modify subsets of strings.

Creating Strings

String

Standard data type: 8 bit character string.

String (Function)

Returns a **String** of multiple characters.

ZString

Standard data type: null terminated 8 bit character string.

WString

Numeric Serialization

MKD

Returns an eight character **String** representation of a **Double**.

MKI

Returns a four character **String** representation of a **Integer**.

MKL

Returns a four character **String** representation of a **Long**.

Standard data type: wide character string.

Wstring (Function)

Returns a **WString** of multiple characters.

Space

Returns a **String** consisting of spaces.

WSpace

Returns a **WString** consisting of spaces.

Len

Returns the length of a string in characters.

Character Conversion

Asc

Returns an **Integer** representation of a character.

Chr

Returns a string of one or more characters from their ASCII **Integer** representation.

WChr

Returns a **WString** of one or more characters from their Unicode **Integer** representation.

Numeric/Boolean to String Conversions

Bin

Returns a binary **String** representation of an integral value.

WBin

Returns a binary **WString** representation of an integral value.

Hex

MKLongInt

Returns an eight character **String** representation of a **LongInt**.

MKS

Returns a four character **String** representation of a **Single**.

MKShort

Returns a two character **String** representation of a **Short**.

CVD

Returns a **Double** representation of an eight character **String**.

CVI

Returns an **Integer** representation of a four character **String**.

CVL

Returns a **Long** representation of a four character **String**.

CVLongInt

Returns a **LongInt** representation of an eight character **String**.

CVS

Returns a **Single** representation of a four character **String**.

CVShort

Returns a **Short** representation of a two character **String**.

Working with Substrings

Left

Returns a substring of the leftmost characters in a string.

Mid (Function)

Returns a substring of a string.

Right

Returns a hexadecimal **String** representation of an integral value.

WHex

Returns a hexadecimal **WString** representation of an integral value.

Oct

Returns an octal **String** representation of an integral value.

WOct

Returns an octal **WString** representation of an integral value.

Str

Returns the **String** representation of numeric value or boolean.

WStr

Returns the **WString** representation of numeric value.

Format

Returns a formatted **String** representation of a **Double**.

String to Numeric Conversions

Val

Returns the **Double** conversion of a numeric string.

ValInt

Returns the **Integer** conversion of a numeric string.

ValLng

Returns the **Long** conversion of a numeric string.

ValUInt

Returns the **UInteger**

Returns a substring of the rightmost characters in a string.

LCase

Returns a copy of a string converted to lowercase alpha characters.

UCase

Returns a copy of a string converted to uppercase alpha characters.

LTrim

Removes surrounding substrings or characters on the left side of a string.

RTrim

Removes surrounding substrings or characters on the right side of a string.

Trim

Removes surrounding substrings or characters on the left and right side of a string.

InStr

Returns the first occurrence of a substring or character within a string.

InStrRev

Returns the last occurrence of a substring or character within a string.

Mid (Statement)

Copies a substring to a substring of a string.

LSet

Left-justifies a string.

RSet

Right-justifies a string.

conversion of a numeric string.

ValULng

Returns the **Ulong** conversion of a numeric string.

Threading Support Functions



Procedures for working with multithreaded applications.

Description

These procedures allow for multithreaded programming. Threads and conditional variables can be created and destroyed, and so-called *mutexes* can be obtained to protect thread-sensitive data.

Threads

Procedures that start and wait for threaded procedures.

Conditional Variables

Procedures that create and signal conditional variables.

Mutexes

Procedures that deal with mutexes.

Threads

Threadcall

Starts a procedure with parameters in a separate thread of execution.

ThreadCreate

Starts a procedure in a separate thread of execution.

Threaddetach

Releases a thread handle without waiting for the thread to finish.

ThreadWait

Waits for a thread to finish and releases the thread handle.

Conditional Variables

CondCreate

Creates a conditional variable.

CondWait

Mutexes

MutexCreate

Creates a mutex.

MutexLock

Acquires a lock on a mutex.

MutexUnlock

Releases a lock on a mutex.

MutexDestroy

Destroys a mutex that is no longer needed.

Pauses execution of a threaded procedure.

CondSignal

Resumes execution of a threaded procedure waiting for a conditional.

CondBroadcast

Resumes all threaded procedures waiting for a conditional.

CondDestroy

Destroys a conditional variable that is no longer needed.

Platform Differences

- These procedures are not supported in DOS.

Statements and procedures that get input from the user.

Description

These statements and procedures allow access to the keyboard buffer and provide ways of getting input from the user.

Reading keys from the keyboard buffer

Procedures that read individual keys from the keyboard buffer.

Reading values from the keyboard buffer

Procedures that read characters and values from the keyboard buffer.

Reading values from the keyboard buffer

Input

Reads values from the keyboard buffer.

Line Input

Reads a line of text from the keyboard buffer.

Input()

Reads a number of characters from the keyboard buffer, file or device.

Winput()

Reads a number of wide characters from the keyboard buffer, file or device.

Reading keys from the keyboard buffer

Inkey

Gets the first key, if any, waiting in the keyboard buffer.

GetKey

Gets and waits for the first key in the keyboard buffer.

Statements and procedures for working with 2D graphics.

Description

The statements and procedures listed here provide ways of drawing to the screen. Image buffers can be created and blitted to the screen using a variety of blending methods. Palette colors can be retrieved or set in graphics modes that support them.

Working with Color

Procedures that control the color used by other drawing procedures.

Drawing to Image Buffers

Procedures that draw shapes and text onto image buffers or to the screen.

Image Buffer Creation

Procedures that create, free and save image buffers.

Blitting Image Buffers

Procedures that draw image buffers onto other image buffers or to the screen.

Working with Color

Color

Sets the foreground and background color to use with the drawing procedures.

Palette

Gets or sets color table information in paletted modes.

RGB

Returns a color value for hi/truicolor modes.

RGBA

Returns a color value including alpha (transparency) for hi/truicolor modes.

Blitting Image Buffers

Put (Graphics)

Blits an image buffer to another image buffer or screen.

Blending Methods

Add

Saturated addition of the source and target components.

Alpha

Blend using a uniform transparency or the image buffer's alpha channel.

And

Combine the source and target

Point

Gets a pixel value from an image buffer or screen.

Drawing to Image Buffers

PSet and PReset

Plots a single pixel on an image buffer or screen.

Line (Graphics)

Plots a line of pixels on an image buffer or screen.

Circle

Plots circles and ellipses on an image buffer or screen.

Draw

Draws in a sequence of commands on an image buffer or screen.

Draw String

Writes text to an image buffer or screen.

Paint

Fills an area with color on an image buffer or screen.

Image Buffer Creation

Get (Graphics)

Creates an image buffer from a portion of another image buffer or screen.

ImageCreate

Creates an image buffer of a certain size and pixel depth.

ImageDestroy

Frees an image buffer resource.

ImageConvertRow

Converts a row of pixels in an image buffer to a different color depth.

components using a bitwise **And Or**

Combine the source and target components using a bitwise **or PSet**

Directly copy pixel colors from the source to the destination.

Trans

Pixels matching the transparent mask color are not blitted.

Custom

Allows a custom blending procedure to be used.

Xor

Combine the source and target components using a bitwise **xor**

ImageInfo

Retrieves useful information about an image buffer

BLoad

Creates an image buffer from a file.

BSave

Saves an image buffer to a file.

User Input Functions



Procedures for working with mice, gaming devices and keyboards.

Description

These procedures provide access to external devices such as keyboards, mice and gamepads.

Mouse and Joystick Input

Procedures that provide state information of the mouse or joystick.

Keyboard Input

Procedures that provide keyboard state information.

Mouse and Joystick Input

GetMouse

Gets button and axis information for the mouse.

SetMouse

Sets position and visibility of the mouse cursor.

GetJoystick

Gets button and axis information for gaming devices.

Stick

Gets axis position for gaming devices.

Strig

Gets button state for gaming devices.

Keyboard Input

MultiKey

Gets key information for the keyboard.

Screen Functions



Statements and procedures that work with the graphics display.

Description

These statements and procedures control the graphics capabilities of the FreeBASIC graphics library. Screen modes can be set with varying resolutions and color depths, window events can be handled, and specific OpenGL procedures can be retrieved.

Working with screen modes

Procedures for setting and retrieving information about screen modes

Working with pages

Procedures that manipulate screen pages.

Working video memory

Procedures that provide direct access to framebuffer memory.

Screen Metrics

Procedures that control the way coordinates are interpreted.

Working with screen modes

ScreenList

Gets the available fullscreen resolutions.

Screen and ScreenRes

Sets a new graphics display mode.

ScreenInfo

Gets information about the system desktop or current display mode.

ScreenControl

Gets or sets internal graphics library settings.

ScreenEvent

Gets system events.

ScreenGLProc

Working video memory

ScreenPtr

Gets the address of the working page's framebuffer.

ScreenLock

Locks the current working page's framebuffer for direct access.

ScreenUnlock

Reverts a previous **ScreenLock** command.

Screen Metrics

View (Graphics)

Sets a clipping region for all drawing and blitting procedures.

Window

Returns the address of an OpenGL procedure.

WindowTitle

Sets the running program's window caption.

Working with pages

Cl

Clears the entire screen or viewport.

ScreenSet

Sets the current work and visible pages.

ScreenCopy and PCopy and Flip

Copies pixel data from one page to another.

ScreenSync

Waits for the vertical refresh of the monitor.

Sets a new coordinate mapping for the current viewport.

PMap

Converts coordinates between physical and view mappings.

Pointcoord

Queries **Draw**'s pen position.

GfxLib - FreeBASIC graphics library overview



GfxLib is the built-in graphics library included in FreeBASIC. As well as re-creating every QuickBASIC graphics command, GfxLib has built-in commands to handle input from the keyboard and mouse. Major contributors of the library are Lillo, CoderJeff and DrV.

The library supports various drivers depending on the platform:

- All:
 - **Null** Does nothing, allows to use graphics functions on in-memory buffers and such, without anything being displayed in a graphics window. ([gfxlib2/gfx_driver_null.c](#))

- Win32:
 - **DirectX** The default selection of FB GfxLib. May not be available on old Windows installations. ([gfxlib2/win32/gfx_driver_ddraw.c](#))
 - **GDI** The "safest" one, available in all Win32 versions. Bug note: broken in FB versions 0.20 to 0.24 (crash), and minor problems 0.18.5, and 0.90.x and 1.xx ("banding effects", try extra SCREENUNLOCK), ([forum discussion: p=106600](#)) ([gfxlib2/win32/gfx_driver_gdi.c](#))
 - **OpenGL** ([gfxlib2/win32/gfx_driver_opengl.c](#))

- Linux & others:
 - **x11** The default on Unix systems ([gfxlib2/unix/gfx_driver_x11.c](#))
 - **OpenGL** (on top of X11) ([gfxlib2/unix/gfx_driver_opengl_x11.c](#))
 - **FBDev** Linux framebuffer device -- fallback in case X11 is disabled ([gfxlib2/linux/gfx_driver_fbdev.c](#))

- DOS:
 - **BIOS** ([gfxlib2/dos/gfx_driver_bios.c](#))

- **ModeX "tuned" 320x240x8bpp VGA mode** ([gfxlib2/dos/gfx_driver_modex.c](#))
- **VESA banked** compatible with very old VESA 1.x implementations ([gfxlib2/dos/gfx_driver_vesa_bnk.c](#))
- **VESA linear** needs VESA version at least 2.0, usually faster than banked VESA ([gfxlib2/dos/gfx_driver_vesa_lin.c](#))
- **VGA** ([gfxlib2/dos/gfx_driver_vga.c](#))
- Bug note: **Palette** doesn't work well ([forum discussion: t=12691 2008](#)) ([forum discussion: t=19980 2012](#))

`ScreenControl1` can be used (`SET_DRIVER_NAME 103`) to override the default driver preferences.

Platform Differences

- In DOS, GfxLib will create and "manage" a mouse arrow if a mouse driver is detected. There is no "official" way to disable this. Also note that the arrow doesn't react to mouse movements while the screen is locked.
- In DOS, Windowing and OpenGL related commands and switches are not available (they exist but do nothing, or return some values with no meaning)
- In DOS, the refresh rate setting is not available (some VESA cards do support it, but FreeBASIC for now doesn't)
- In DOS, the resolution must match one supported by the graphics card. GfxLib will try to find an appropriate mode from VGA modes, ModeX or VESA, preferring VESA LFB interface if available, or banked VESA otherwise. Unsupported resolutions may currently crash the program (if you fail to check `SCREENPTR` for ZERO before using it), though in future GfxLib may try to find a close match instead. For optimal compatibility you should support "safe" resolutions like 640x480 and 800x600, and maybe 1024x768. There are various additional modes like 768x576 around, but they are vendor specific and lacking on many other cards. Also modes 1024x768 and above are not available on older cards and laptops.
- It has been observed that `SCREEN` and `SCREENRES` may fail

to clear the screen in DOS, actually this is probably a BIOS bug that GfxLib currently doesn't work around.

Differences from QB

- Graphics support was internally redesigned. QB used VGA graphics modes, and wrote directly into the VGA RAM. Multiple pages were available as long as the card supported them. FB uses backbuffers, one per defined page, and copies them to the video RAM (VGA (DOS), VESA (DOS), DirectX (Win32), ...) in the background. Graphics commands do work as they used to in QB, but a few notable differences are present:
 - The background screen updating eats a considerable amount of CPU performance.
 - There is a thread (Win32 and Linux) or ISR (DOS, uses the PIT) active for this.
 - Mixing FB's graphics support with low-level screen accesses (VGA) is not supported, even in DOS. However direct screen memory access is possible using `ScreenPtr` and `ScreenLock` and is fully portable. In DOS VGA and VESA are still available, but can't be mixed with FB's graphics support.

See also

- [GFX Functions Index](#)
- `screen` The QB-like way to set graphics mode
- `ScreenRes` More flexible alternative to `screen`
- `ScreenList` Check display modes available for FB GfxLib to use
- `ScreenControl` Select driver and more
- `ScreenLock`
- `ScreenUnlock`
- `ScreenPtr` Semi-low level access
- `ScreenSet`
- `ScreenCopy`

- `ScreenInfo`
- `ScreenGLProc`
- `Internal pixel formats`

DOS Keyboard Scancodes



Listing of keyboard scancodes.

Description

Here follows a list of hardware keyboard scancodes accepted by the `INKEY` function. These are equal to DOS scancodes, and are guaranteed to be recognized on all platforms.

These constants are also defined in the `fbgfx.bi` include file you can use in your programs. If you are using the *lang fb* dialect then everything in `fbgfx.bi` is enclosed in the `FB` namespace. To use these constants in *lang fb* either prepend "FB." to the constant name, or put "Using FB" after the line.

The hexadecimal code is not required and provided only for reference

SC_ESCAPE	&h01
SC_1	&h02
SC_2	&h03
SC_3	&h04
SC_4	&h05
SC_5	&h06
SC_6	&h07
SC_7	&h08
SC_8	&h09
SC_9	&h0A
SC_0	&h0B
SC_MINUS	&h0C
SC_EQUALS	&h0D
SC_BACKSPACE	&h0E
SC_TAB	&h0F
SC_Q	&h10
SC_W	&h11
SC_E	&h12
SC_R	&h13
SC_T	&h14
SC_Y	&h15

SC_U	&h16
SC_I	&h17
SC_O	&h18
SC_P	&h19
SC_LEFTBRACKET	&h1A
SC_RIGHTBRACKET	&h1B
SC_ENTER	&h1C
SC_CONTROL	&h1D
SC_A	&h1E
SC_S	&h1F
SC_D	&h20
SC_F	&h21
SC_G	&h22
SC_H	&h23
SC_J	&h24
SC_K	&h25
SC_L	&h26
SC_SEMICOLON	&h27
SC_QUOTE	&h28
SC_TILDE	&h29
SC_LSHIFT	&h2A
SC_BACKSLASH	&h2B
SC_Z	&h2C
SC_X	&h2D
SC_C	&h2E
SC_V	&h2F
SC_B	&h30
SC_N	&h31
SC_M	&h32
SC_COMMA	&h33
SC_PERIOD	&h34
SC_SLASH	&h35
SC_RSHIFT	&h36
SC_MULTIPLY	&h37
SC_ALT	&h38
SC_SPACE	&h39
SC_CAPSLOCK	&h3A
SC_F1	&h3B
SC_F2	&h3C

SC_F3	&h3D
SC_F4	&h3E
SC_F5	&h3F
SC_F6	&h40
SC_F7	&h41
SC_F8	&h42
SC_F9	&h43
SC_F10	&h44
SC_NUMLOCK	&h45
SC_SCROLLLOCK	&h46
SC_HOME	&h47
SC_UP	&h48
SC_PAGEUP	&h49
SC_LEFT	&h4B
SC_RIGHT	&h4D
SC_PLUS	&h4E
SC_END	&h4F
SC_DOWN	&h50
SC_PAGEDOWN	&h51
SC_INSERT	&h52
SC_DELETE	&h53
SC_F11	&h57
SC_F12	&h58

' ' Extra scancodes not compatible with DOS scancoc

SC_LWIN	&h7D
SC_RWIN	&h7E
SC_MENU	&h7F

See also

- [MultiKey](#)

Default Palettes



Default color values for FreeBASIC graphics and text screen modes.

FreeBASIC initializes the palette indexes with the colors in the tables below. In graphics mode, these can be changed using the `palette` statement. There is also a console mode.

Screen mode 1

4 colors: Black and white, and two others

Screen modes 2, 10 and 11

Monochromatic: black and white.

Screen modes 7, 8, 9, 12, and Console

Two sets of 8 colors: normal and intense (bright)

Screen 13 and 8-bit modes

Multiple color and grayscale bands

Screen mode 1



Value	Name
0	black
1	cyan
2	magenta
3	white

Screen modes 2, 10 and 11



Value	Name

Screen 13 and 8-bit

Screen 12 color bar

Colors 0 through 15

Grayscale band

Colors 16 through 31

Brightness/saturation band

3 bands of decreasing saturation, decreasing saturation and ending at blue.

Name	HB/HS	HB/MS
blue	32	56

0	black
1	white

magenta	36	60
red	40	64
yellow	44	58
green	48	72
cyan	52	76

Screen modes 7, 8, 9, 12, and Console



Black band
Colors 248 through 2

Normal Value	Normal Name	Intense Value	Intense Name
0	black	8	dark grey
1	blue	9	bright blue
2	green	10	bright green
3	cyan	11	bright cyan
4	red	12	bright red
5	pink	13	bright pink
6	yellow	14	bright yellow
7	grey	15	white

FreeBASIC programmer's guide.

Work in Progress: New pages created for this guide should use the ProPg prefix.*

Getting Started

[Hello World](#)

[FreeBASIC Primer #1](#)

Source Files

[Source Files \(.bas\)](#)

[Header Files \(.bi\)](#)

[Using Prebuilt Libraries](#)

Lexical Conventions

[Comments](#)

[Identifier Rules](#)

[Literals](#)

[Labels](#)

[Line continuation](#)

Variables and Datatypes

[Constants and Enumerations](#)

[Numeric Types](#)

[Strings \(string, zstring, and wstring\)](#)

[Coercion and Conversion](#)

[Constants](#)

[Variables](#)

Arrays

[Overview](#)

[Fixed-length Arrays](#)

[Variable-length Arrays](#)

[Array Indexing](#)

Statements and Expressions

[Assignments](#)

[Operators List](#)

[Operator Precedence](#)

[Control Flow Statements](#)

Procedures

[Procedures Overview](#)

[Passing Arguments to](#)

[Procedures](#)

[Returning a Value](#)

[Procedure Scopes](#)

[Calling Conventions](#)

[Recursion](#)

[Constructors and Destructors](#)

[Pointers to Procedures](#)

[Variable Arguments](#)

Making Binaries

[Executables](#)

[Static Libraries](#)

[Shared Libraries \(DLLs\)](#)

[Profiling](#)

Preprocessor

[Overview](#)

[Conditional Compilation](#)

[Macros](#)

Other Topics

Passing Arrays to Procedures

Pointers

Overview

Pointer Arithmetic

Declarations

Implicit Declarations

Initialization

Storage Classes

Variable Lifetime

Variable Scope

Namespaces

Variable and Procedure

Linkage

User Defined Types

Overview

Type Aliases

Temporary Types

Constructors and Destructors

Member Procedures

Properties

Member Access Rights

Operator Overloading

Iterators

New and Delete

Types as Objects

(And topics that need to get placed elsewhere)

ASCII

Date Serials

Radians

FreeBASIC GfxLib overview

Internal Graphics Formats

External Graphics File Formats

Inline Asm

Error Handling

Intrinsic Defines

C Standard Library Functions

File I/O in FreeBASIC

NOTE: Existing CatPg pages should be recreated as ProPg pages providing a general overview to the grouping of keywords.

Hello World



This example is a classic in any programming language.

More as a sanity check than anything else, a good place to start with an programming language is to try a very simple program to test that the compiler is installed correctly and that a valid executable can be made.

Open up any editor capable of saving text files and type in the following source code:

```
Print "Hello World"
```

Save the file with a '.bas' extension. For example 'hello.bas'

From a command prompt or shell in the directory where 'hello.bas' was saved, type the following command:

```
fbc hello.bas
```

Depending on the operating system, this should create an executable file in the same directory as 'hello.bas'. It might be named 'hello.exe' or './hello', for example.

Run the executable, and we should have the following output:

```
Hello World
```

See also

- [Freebasic FAQ](#)
- [Main Features](#)
- [Requirements](#)
- [Installing](#)
- [Running](#)

This primer is intended for beginning beginners, for those who are just starting to program and using FreeBASIC to do it.

Learning the language

Learning a programming language means learning the words to write and what they mean when they are written. We don't need to learn them all at once, but a few important words that do something will help us get started. Here we concentrate on these keywords:

- `Dim`
- `Print`
- `Input`
- `For...Next`
- `If...Then`
- `Do...Loop`

Hello World!

No beginner's reference is complete without this example.

```
Print "Hello World!"
```

The text between the pair of double quotes is a literal string. The `Print` statement outputs text to the display. If you can edit, compile, and execute this example your way.

Using a Variable to Store Data

Sometimes in a program we will want to store some information and then use it later. To store something in memory we use a variable. FreeBASIC variables are of some specific type, like a number or a string. We usually declare a variable name and specify what type of information we want to store.

```
Dim text As String
text = "Hello World!"
Print text
```

We are using `Dim` to let the compiler know that we want to use a variable in our program and that we will be putting `String` data in it. We then assign a value to the variable. Finally, we use `Print` to output it to the display.

Using a Variable in an Expression

An *expression* is a generic term for describing a part of the source code that is evaluated. After an expression is evaluated, we can then do something (copy) it to a variable.

```
Dim a As String, b As String, text As String
a = "Hello"
b = "World"
text = a + " " + b + "!"
Print text
```

We are assigning the variables `a` and `b` with some data. We are then using `a` and `b` in an expression which is then assigned to `text`. Finally, we output the value of `text` to the display.

Getting Input from the User

Often, we have no idea what data is needed for a program unless the user provides it. We can't put it in our source code since we won't know what it is until the user runs the program and tells us what it is.

```
Dim answer As String
Input "Type something and press enter:", answer
Print "You typed: "; answer; ""
```

Here the **Input** statement will first, output some information to the display the user to give the program some data. In this example, we just output exactly what the user typed in.

Doing Some Math

Variables and expressions are not just limited to strings. Most early languages strings very well if at all. Writing mathematical expressions is similar to what is written with pencil and paper.

```
Dim a As Integer, b As Integer, c As Integer

a = 5
b = 7
c = a + b

Print "a = "; a
Print "b = "; b
Print "a + b = "; c
```

We are assigning values to the variables *a*, *b* and *c*. We are using **Integer** data type. An integer can be positive or negative, but not have any fractional part.

Doing Some Math with Input

This is similar to the previous example, except we will let the user choose the numbers they are going to add together.

```
Dim a As Integer, b As Integer, r As Integer
Input "Enter a number:", a
Input "Enter another number:", b

r = a + b
Print "The sum of the numbers is "; r
```

`Dim` lets the compiler know which variable names we want to use and hold `Integer` data. We are using `Input` to get the numbers from the user and `Print` to display the results.

Doing More Math with Input

Numeric variables are not limited to just integers. We can also use `Single` precision data types which can represent fractions. In this example we get input from the user to convert a weight in pounds to kilograms.

```
Dim lb As Single, kg As Single
Input "Enter a weight in pounds:", lb

kg = lb * 0.454
Print lb; " lb. is equal to "; kg; " kg"
```

Repeating Statements

Using `For...Next` statement we can tell the program to do something a certain number of times. For example let's say we wanted to add up all the numbers from 1 to 100.

```
Dim total As Integer
Dim number As Integer
total = 0
For number = 1 To 100
    total = total + number
Next
Print "The sum of number from 1 to 100 is "; total
```

Making a Decision

A program can choose which statements to execute using a condition.

If...Then. We can use the value of a variable or the result of an expression to determine if a condition should, or should not, execute one or more statements.

```
Dim number As Integer
Input "Enter a number : ", number
Print "Your number is ";
If number < 0 Then
    Print "negative"
ElseIf number > 0 Then
    Print "positive"
Else
    Print "zero"
End If
```

After getting a number from the user, we are going to output a word (zero) based on which condition matches the statement.

Repeating Statements (Again)

Here we will use another looping structure **Do...Loop** to repeat some statements. How do we know when the program knows to stop repeating the statements? We will use **If...Then** decision when to get out of the loop.

```
Dim total As Single, count As Single, number As Single
Dim text As String

Print "This program will calculate the sum and average of a
Print "list of numbers. Enter an empty value to end the list."
Print

Do
    Input "Enter a number : ", text
    If text = "" Then
        Exit Do
    End If
```

```
count = count + 1
total = total + Val(text)
```

Loop

```
Print
Print "You entered "; count; " numbers"
Print "The sum is "; total
If count <> 0 Then
    Print "The average is "; total / count
End If
```

See also

- Dim
- (Print | ?)
- Input
- For...Next
- If...Then
- Do...Loop

Source Files (.bas)



Text files read by FreeBASIC and compiled into executable code.

A source file is a text file that contains FreeBASIC language statements. just one source file or possibly hundreds. Source files are read by the cc code. Object code is then linked to create an executable or can be store

FreeBASIC by default, automatically takes care of compiling sources an executables, so normally it is possible to make an executable program k source files on the fbc command line. For example, assuming we had th made a program, we could create an executable for the program by runi on a command line as follows:

```
fbc myprog.bas tools.bas funcs.bas
```

Unicode support

- Besides ASCII files with Unicode escape sequences (\u), Free 16LE, UTF-16BE, UTF-32LE and UTF-32BE source (.bas) or h mixed with other sources/headers in the same project (also wit
- Literal strings can be typed in the original non-Latin alphabet, j one of the Unicode formats listed above.

Implicit main()

Some languages require a special `main()` procedure be defined as an define the first statements that will be executed when the program sta statements in module level code and normally the first source file pas: be used as the "main" module. The main module can be explicitly nan command line, where *filename* is the name of the main module withou

```
' ' sample.bas
Declare Sub ShowHelp()

' ' This next line is the first executable stat
If Command(1) = "" Then
    ShowHelp
```

```
End 0
End If

Sub ShowHelp()
    Print "no options specified."
End Sub
```

Header Files

A header file is a special kind of source file that typically only contains extension. See **[Header Files \(.bi\)](#)**.

See also

- **[fbc command-line](#)**
- **[Header Files \(.bi\)](#)**

Header Files (.bi)



Provides an interface for a module.

A header file is a special kind of source file that typically only contains preprocessor statements, defines, declarations, prototypes, constants, enumerations, and macros. However, a header file can contain any valid source code if it is not preprocessed. What makes them different from other module (.bas) source files, is that they are included directly, they are included by another source file (module or header) using the preprocessor directive. All compiled libraries typically have one or more header files that can be included in another source file and will introduce to the compiler all the procedures usable in a particular library.

FreeBASIC Header Files

Some of the keywords, constants, and procedures documented in this manual are not normally available when compiling a source code unless a specific header file is included in the source first.

- `datetime.bi`
- `dir.bi`
- `fbgfx.bi`
- `file.bi`
- `string.bi`
- `vbcompat.bi`

Case Sensitivity

Although the FreeBASIC language itself is not case-sensitive, the file names used in the source code might be. If a header file can not be found, check that FreeBASIC is running from the correct location and ensure that name of both the directory and file specified in the `#include` statement is using the correct upper and lower case.

Path Separators

FreeBASIC will automatically switch backslash (\) and forward slash (/) as needed for a given platform. This allows source code to be easily cross-platform.

Including a header only once

It is common that header files need to `#include` other header files to c FreeBASIC offers three methods for guarding against including a header once.

- `#ifndef` guards in the header file
- `#include once` where the file is included
- `#pragma once` in the header file itself

#ifndef guards in the header file

The use of `#ifndef` and `#define` is a common practice in nearly any language preprocessing. The first time a file is included, a unique symbol is defined. If the same header file is included, the definition of the symbol is checked, and if it is defined, the contents of the header file are skipped.

```
' ' header.bi
#ifndef __HEADER_BI__
#define __HEADER_BI__

#print These statements will only be included once
#print even though header.bi might be included more
#print than once in the same source file.

#endif
```

#include once

At the point in the source code where the header file is included, the compiler specifier of the `#include` directive can tell the compiler to only include the file one time.

```
' ' header.bi
#include once "fbgfx.bi"
```

```
' ' module.bas
```

```
#include once "fbgfx.bi"  
#include once "header.bi"
```

#pragma once

#pragma once can be used in a header file to indicate that the header file is included only once.

```
' ' header.bi  
#pragma once  
#print This header will only ever be included once
```

See also

- **Source Files (.bas)**
- **Header Files Index**

Using Prebuilt Libraries



FreeBASIC is distributed with many headers for common or popular libraries. These headers allow a programmer to use functions available in these existing shared libraries (DLLs).

The libraries themselves are not distributed with FreeBASIC, but most can be downloaded from the web and readily installed. Some other libraries may need to be first compiled from sources to be used. Please see the documentation for a specific library on how to configure, install, and use them.

Some static or shared libraries (DLLs) may be already present on the system. They might be part of FreeBASIC itself or the operating system.

Although many headers can be used on any of the platforms supported by FreeBASIC, some headers are platform specific and will not be usable on all platforms.

FreeBASIC headers

There are a few headers that are specific to FreeBASIC and expose some functions that are otherwise not available:

- `datetime.bi` - Declarations for `DateSerial`, `DateValue`, `IsDate`, `Year`, `Day`, `Weekday`, `TimeSerial`, `TimeValue`, `Hour`, `Minute`, `Second`, `Now`, `Month`, `DatePart`, `DateDiff`, `MonthName`, `WeekdayName`
- `dir.bi` - Constants to be used with `Dir`
- `fbgfx.bi` - Additional constants and structures to be used with commands such as `MultiKey`, `ScreenControl`, and `ScreenEvent`, `ImageCreate`.
- `file.bi` - Declarations for `FileCopy`, `FileAttr`, `FileLen`, `FileExist`, `FileDateTime`
- `string.bi` - Declarations for `Format`
- `vbcompat.bi` - Includes `datetime.bi`, `dir.bi`, `file.bi`, and `string.bi` plus additional constants compatible with Microsoft Visual Basic.

C Runtime (CRT)

Where possible cross-platform compatible headers have been provided for the Windows runtime (CRT). For example,

```
#include once "crt.bi"
printf( !"Hello World\n" )
```

To include a specific CRT header, prefix the name of the header file with "crt/" for example:

```
#include once "crt/stdio.bi"
Dim f As FILE Ptr
f = fopen("somefile.txt", "w")
fprintf( f, "Hello File\n")
fclose( f )
```

Windows API

Many (many) headers for the Windows API are available for inclusion in FreeBASIC source code. In most cases the only include file needed is "windows.bi". For example,

```
#include once "windows.bi"
MessageBox( null, "Hello World", "FreeBASIC", MB_C
```

To include a specific Windows API header, prefix the name of the header with "win/" for example:

```
#include once "win/ddraw.bi"
```

Browse the "inc/win/" directory where FreeBASIC was installed to see what Windows API headers are available.

Other Headers Provided

Browse the "inc/" directory located where FreeBASIC was installed to see what headers are available. It is possible that headers might be available for a library you

use. Some headers are located in "inc/" and others might be located in a subdirectory. To include headers located in a subdirectory of "inc/", prefix the header with the name of the directory where it is located. For example:

```
' ' located at inc/curl.bi  
#include once "curl.bi"  
  
' ' located at inc/GL/gl.bi  
#include once "GL/gl.bi"
```

Requirements for Using Prebuilt Static Libraries

- The source code must include the appropriate headers using #include
- The static library must be linked at compile time by using either the -l option on the command line or by using the -L option on the command line to specify the name of the library.

Requirements for Using Prebuilt Shared Libraries

- The source code must include the appropriate headers using #include
- The shared library (.DLL) must be present on the host computer where the compiled program will run.

Comments



Comments are regions of text that the compiler will ignore but may contain information that is useful to the programmer. One exception are metacommands which appear in certain types of comments.

Single Line comments

The single quote character (') may be used to indicate a comment and may appear after other keywords on a source line. The rest of the statement will be ignored as comment.

```
' comment text
```

The comment statement: Rem

A source code statement beginning with **Rem** indicates that the rest of the statement is comment and will not be compiled. **Rem** behavior is the same as above and **Rem** must be the first keyword in the statement.

```
Rem comment
```

Multi-line comments

Multi-line comments are marked with the tokens `/'` and `'/`. All text between these markers is considered comment text and is not compiled.

Multi-line comments can span several lines, and can also be used in multi-line statements. After the end of the comment, the statement will continue as normal (even if the comment crosses line breaks).

```
/' Multi-line  
comment '/  
  
Print "Hello" /' embedded comment' / " world"
```

Note: If FreeBASIC encounters a close-comment marker while it's not comment, it will treat it as a normal single-line comment due to the sir

Nested Comments

A multi-line comment can contain other multi-line comments inside it. comment has its own open- and close-comment markers.

```
/'  
  This is a comment.  
  /'  
    This is a comment inside a comment  
  '/  
    This Is a comment.  
  /'
```

A multi-line comment can contain unlimited levels of nested comment: will continue to parse the multi-line comment for more markers until the close-comment markers reaches the number of open-comment markers has closed all the comments it has opened.

Comments after line continuation

A single-line comment may appear after the line continuation character multi-line statement. FreeBASIC does not parse the text after the line character, though, so you can't open multi-line comments after them.

```
Print _ ' line  
  "This is part of the previous line's statement"
```

Metacommands

Metacommands, such as `$Static` and `$Include`, can be placed in single-line comments. The `$` sign and the keyword must be the first two things in

not including white space.

```
Rem compile With -lang fblite Or qb  
  
#lang "fblite"  
  
Rem $Static  
' $include: 'vbcompat.bi'
```

Single-line comment parsing

When you make a single-line comment, FreeBASIC will parse the comment and check for a metacommand. If it finds a multi-line comment, it will treat it as a multi-line comment and continue parsing the single-line comment after the close-comment marker.

If you want to prevent FreeBASIC parsing the single-line comment, put a single quote ('), at the start of the comment. FreeBASIC will treat the line, including multi-line comment markers and metacommands, as ordinary text and will ignore it. Other words encountered in a comment will also stop parsing.

- *Note: As of version 0.21.0, this will not longer apply in the FreeBASIC dialect, and multi-line comment markers will be completely ignored inside single-line comments*

```
' $static <-- will not get parsed  
' this multiline comment marker ("/'") will be ignored  
Print "This line is not a comment."
```

Example

```
/' this is a multi line  
comment as a header of  
this example '/
```

```
Rem This Is a Single Line comment

'this is a single line comment

Dim a As Integer 'comment following a statement

Dim b As '/' can comment in here also '/' Integer

#if 0
    before version 0.16, This was the
    only way of commenting Out sections
    With multiple lines of code.
#endif
```

See also

- Rem

Naming conventions for FreeBASIC symbols.

Description

An identifier is a symbolic name which uniquely identifies a **variable**, **Type**, **Union**, **Enum**, **Function**, **Sub**, Or **Property**, within its **scope** or **Namespace**.

Identifiers may contain only uppercase and lowercase Latin character a-z and A-Z), digits (0-9), and the underscore character (_). The first character of an identifier must be a letter or underscore, not a digit.

Identifiers are case-insensitive: F00 and f00 (and all other permutation of uppercase and lowercase) refer to the same symbol.

In the *-lang qb* and *-lang fblite* dialects, identifiers may have a type suffix at the end indicating one of the standard data types:

- % for **Integer**
- & for **Long**
- ! for **Single**
- # for **Double**
- \$ for **String**

The use of these symbols is generally discouraged in and is not allowed in the *-lang fb* dialect (the default).

The alternative is to be explicit - for example, `Dim As Integer foo` Or `Dim foo As Integer` instead of `Dim foo%`.

In the *-lang qb* and *-lang fblite* dialects, identifiers may contain one or more periods (.

Dialect Differences

- Periods in symbol names are only supported in the *-lang qb* and *-lang fblite* dialects.

Differences from QB

- Support for the underscore character (`_`) in symbol names is new to FreeBASIC.

See also

- **Variables**

Literals



Non-variable compile-time string, numeric values and boolean values.

Literals are numbers, strings of characters or boolean truths specified in the source code. Literal values may be used by assigning them to a variable, constant, passing them to a procedure, or using them in an expression.

Numeric literals come in two forms - integer and floating-point.

Integer Literals

Decimal

Decimal digits (0 1 2 3 4 5 6 7 8 9).

Note: to get negative values, a "-" sign (Operator - (Negate)) can be before a numeric literal

```
Dim x As Integer = 123456
Dim b As Byte = -128
```

Hexadecimal

"&H;", followed by hexadecimal digits (0 1 2 3 4 5 6 7 8 9 A B C D E F).

```
Dim x As Integer = &h1E240
Dim b As Byte = &H80
```

Octal

"&o;" (O as in "Octal"), followed by octal digits (0 1 2 3 4 5 6 7).

```
Dim x As Integer = &0361100
Dim b As Byte = &0200
```

Binary

"&B;", followed by binary digits (0 1)

```
Dim x As Integer = &B11110001001000000  
Dim b As Byte = &B10000000
```

Integer size suffixes

If an integer literal suffix is not given, the number field size required to literal is automatically calculated. Specifying a size suffix guarantees the compiler will consider a number as a specific integer size.

Integer literals ending with:

- "%", are considered as signed 32/64 (depending on platform) bit integers. (**Integer**)
- "L", "&", are considered as signed 32 bit long integers. (**Long**)
- "U", are considered as unsigned 32/64 (depending on platform) integers. (**UInteger**)
- "UL", are considered as unsigned 32 bit integers. (**Ulong**)
- "LL", are considered as signed 64 bit integers. (**LongInt**)
- "ULL", are considered as unsigned 64 bit integers. (**ULongInt**)

The prefixes, suffixes, and hexadecimal letter digits are all case-insensitive.

```
Dim a As Long = 123L  
Dim b As UInteger = &h1234u  
Dim c As LongInt = 76543LL  
Dim d As ULongInt = &b1010101ULL
```

Floating Point Literals

Floating point numbers are specified in decimal digits, may be positive or negative, have a fractional portion, and optionally an exponent. The format is:

floating point literal is as follows:

```
number[.[fraction]][((D|E) [+|-] exponent)|(D|E)][suffix]  
Or  
.fraction[((D|E) [+|-] exponent)|(D|E)][suffix]
```

By default, floating point numbers that do not have either an exponent are considered as a double precision floating point value, except in the dialect, where numbers of 7 digits or fewer are considered to be single precision.

```
Dim a As Double = 123.456  
Dim b As Double = -123.0
```

The letter "D" or "E", placed after the number/fraction part, allows the number to be given an exponent. The exponent may be specified as either positive or negative with a plus ("+") or minus ("-") sign. Exponents that do not have a sign are positive.

An exponent is not required after the letter, so the letter can be used just to specify the type. "D" specifies a double-precision floating-point number and "E" specifies a floating-point number using the default precision. When the letter is used on its own in combination with a suffix (see below) the type of the suffix overrules the type specified by the letter.

```
Dim a As Double = -123.0d  
Dim b As Double = -123e  
Dim c As Double = 743.1e+13  
Dim d As Double = 743.1D-13  
Dim e As Double = 743.1E13  
Dim f As Single = 743D!
```

A suffix of "!" or "F" on a number specifies a single precision (32 bit) floating point value. A suffix of "#" specifies a double precision float. Note that the letter suffixes and exponent specifiers are all case-insensitive.

```
Dim a As Single = 3.1!  
Dim b As Single = -123.456e-7f  
Dim c As Double = 0#  
Dim d As Double = 3.141592653589e3#
```

String Literals

String literals are a sequence of characters contained between two double quotes. The sequence of characters escaped or non-escaped.

Double quotes can be specified in the string literal by using two double quotes together.

```
Print "Hello World!"  
Print "That's right!"  
Print "See the ""word"" contained in double quotes"
```

String literals can contain escape sequences if the string literal is prefixed by the `!` operator (Escaped String Literal). See [Escape Sequences](#) for a list of accepted escape sequences.

```
Print !"Hello\nWorld!"
```

By default, string literals are non-escaped unless `Option Escape` was used in the source in which case all string literals following are by default escaped.

A string may be explicitly specified as non-escaped when prefixed by the `$` operator (Non-Escaped String Literal).

```
Print $"C:\temp"
```

Besides ASCII files with Unicode escape sequences (`\u`), FreeBASIC

UTF-8, UTF-16LE, UTF-16BE, UTF-32LE and UTF-32BE source files
unicode characters directly in the string literal.

Boolean Literals

The boolean type has two values, represented by literals `True` and `False`

```
Dim a As Boolean = False  
Dim b As Boolean = True
```

See also

- `TypeOf`
- `#define`
- `Const`
- `Standard Data Types`
- `Table with variable types overview, limits and suffixes`

Labels



Defines a location in a program.

Syntax

```
symbolName :  
or  
literalNumber
```

Description

Defines a place in a program where **Goto** or **GoSub** can jump to.

A label can be a positive integer line number or a *symbolName*. In both cases, the label must end with a colon (:) character.

Example

```
' ' Compile with -lang fblite or qb  
  
#lang "fblite"  
  
beginning:  
3 Print "Hello World!"  
Goto beginning
```

```
' ' compile with -lang qb  
  
'$lang: "qb"  
  
' ' Labels can be used to "bookmark" DATA blocks, a  
Read a,b,c  
Restore here  
Read d,e  
Print a,b,c,d,e
```

```
Data 1, 2, 3, 4, 5  
here:  
Data 6, 7, 8
```

Output:

```
1, 2, 3, 6, 7
```

Dialect Differences

- Line numbers with decimals is available only in the *-lang qb* di

Differences from QB

- None if compiled in the *-lang qb* dialect.

See also

- [GoSub](#)
- [Goto](#)

Line continuation



A single _ (underscore) character at the end of a line of code tells the compiler to be spread across multiple lines in the input file, which can be a

```
' This Dim statement is spread across multiple lines
Dim myvariable _
As Integer
```

This is often used to make very long lines of code easier to read, for example

```
' Here's an example:
Declare Sub drawRectangle( ByVal x As Integer, ByVal y As Integer,
                           ByVal w As Integer, ByVal h As Integer )
' which can also be written as:
Declare Sub drawRectangle( ByVal x As Integer, ByVal y As Integer,
                           ByVal w As Integer, ByVal h As Integer )
' or:
Declare Sub drawRectangle _
    ( _
      ByVal x As Integer, _
      ByVal y As Integer, _
      ByVal w As Integer, _
      ByVal h As Integer _
    )
' (or any other formatting you like)
```

The _ line continuation character can be inserted at pretty much any point

Be careful when adding the _ line continuation character right behind an otherwise it would be treated as part of the identifier or keyword.

```
' ' Declare variable "a_"  
' ' (no line continuation happening, because the '_'  
' ' the "a_" identifier)  
Dim As Integer a_  
  
' ' Declare variable "a" and initialize to value 5  
' ' (line continuation happening, because the '_' cha  
' ' was separated from the identifier "a" with a spac  
Dim As Integer a _  
= 5
```

Warning: When an erroneous code line is spread over a multiple lines to the last line of the block.

Coercion of Numeric Data Types in Expressions.

When two different data types are used in a binary operation, like + (Addition) or = (Assignment), the smaller data type is automatically promoted to the larger data type regardless of the order in which the arguments are given.

Promotions are as follows:

- where both arguments are each one of byte, ubyte, short, ushort, or integer: the smaller sized argument is promoted to have the same size as the larger sized argument.
- where one of the arguments is longint or ulongint, and the other argument is of any integer type, the smaller sized argument is promoted to have the same size as the larger sized argument.
- where one of the arguments is a single or a double, both arguments are converted and/or promoted to double

All unsigned integer types are handling like signed integer types for the purpose of promotion, and the most significant bit is extended (sign extension).

Conversion of Numeric Data Types

A type conversion will occur implicitly when an expression or variable is assigned, passed as a parameter to a procedure, or returned as a result from a procedure. Conversions may also be explicit when using CAST or one of the built-in conversion functions.

Integer To Integer, any combination of Signed and Unsigned

- Any integer type to a smaller integer type: least significant bits are retained
- Any integer type to a larger integer type: sign extended to fill most significant bits

Integer to Single or Double

- Possible loss of precision

Double to Single

- Possible loss of precision
- If the value of the Double exceeds the range of a Single result is +/- INF

Double or Single to Integer

- Possible loss of precision
- If the value of the floating point number exceeds the range of the target type are results are undefined. A run-time error is not raised.

See also

- **Standard Data Types**
- **Variable Types**
- **Casting and Conversion Functions**

Description

Constants are numbers which cannot be changed after they are defined. They all mean the same number.

In FreeBASIC, a constant definition differs from a variable definition by

Such constants are then available globally, meaning that once defined to a constant anywhere in your program.

After being defined with the `const` command, constants cannot be altered. If you attempt to change a constant, an error message will result upon code compilation.

Example

```
Declare Sub PrintConstants ()

Const FirstNumber = 1
Const SecondNumber = 2
Const FirstString = "First string."

Print FirstNumber, SecondNumber 'This will print 1 2
Print FirstString 'This will print First string.

PrintConstants ()

Sub PrintConstants ()
    Print FirstNumber, SecondNumber 'This will also print 1 2
    Print FirstString 'This will also print First string.
End Sub
```

See also

- **Const**
- **Enum**

Symbols representing data in memory.

Description

Variables are name symbols which can be manipulated. They are decomposed of letters, numbers, and character "_". These reference name symbols because such symbols are part of the FreeBASIC program and contain spaces. See *Identifier Rules*.

In FreeBASIC, variables can be defined using the `Dim` statement.

Variables are available for later access depending on where and how is given. Depending on the **scope** of a variable, a defined variable can be used in a program, within a procedure, through an entire module, or through *Variable Scope*.

Variables are also made available when they are passed as parameter or `Sub`.

After a variable is declared with the `Dim` statement, they can be assigned in expressions wherever their **Standard Data Type** is similar. Sometimes they are converted to other data types before being used in expressions, or passed. See *Coercion and Conversion*.

Example

```
' compile with -lang qb or fblite
'$lang: "qb"

Declare Sub PrintConstants()

Dim FirstNumber As Integer
Dim Shared SecondNumber As Integer

FirstNumber = 1
```

```
SecondNumber = 2

PrintConstants ()
Print FirstNumber, SecondNumber, ThirdNumber 'This

Sub PrintConstants ()
    Dim ThirdNumber As Integer
    ThirdNumber = 3
    Print FirstNumber, SecondNumber, ThirdNumber '
End Sub
```

See also

- **Coercion and Conversion**
- **Dim**
- **Identifier Rules**
- **Variable Scope**

Multi-dimensional container types.

Overview

Arrays are special kinds of **variables** which act as containers for a number of *elements*. An array can store elements of any type, and all of its elements are accessed--read from or written to--through an **Integer** *position* in the array. Arrays have lengths, or *sizes*, which are equal to the number of elements they are storing at any given time. *Fixed-length* arrays have constant sizes throughout their lifetimes, while the sizes of *variable-length* arrays can change dynamically.

Elements and positions

The values that an array stores are its elements. Each element of an array is accessed by its *position*, which is an **Integer** value ranging from the array's *lower bound* to its *upper bound*, inclusive. These positions are used to access individual elements in the array. The `ArrayElementAt` method takes a position and returns a reference to the element at that position. The `ArrayElementAt` method throws an `ArgumentOutOfRangeException` if the position is greater than or equal to its lower bound, and less than or equal to its upper bound.

```
' Create an array of 3 elements all having the value 0.0
Dim array(1 To 3) As Single

' Assign a value to the first element.
array(1) = 1.2

' Output the values of all the elements ("1.2 0 0")
For position As Integer = 1 To 3
    Print array(position)
Next
```

Sizes and bounds

The size of an array is equal to the number of elements it stores at any given time.

have a size of zero (0), meaning it's not storing any values at the moment. If the size is greater than zero, that many elements are being stored. An array's size is the difference between its upper and lower bounds, or `UBound(array) - LBound(array) + 1`.

The lower and upper bounds not only determine the size of an array, but also the positions of individual elements. For example, an array with lower bound 0 and upper bound 4 stores five (5) elements, the first element being at position 0, the last at position 4. The lower and upper bounds may be specified when the array is declared, or, for some arrays, they may be specified later. An array's lower and upper bounds can be retrieved using `LBound(array)` and `UBound(array)`.

When creating or resizing an array, if a lower bound is not specified it defaults to 0.

```
' Declares and initializes an array of four integers
Dim array(3) As Integer = { 10, 20, 30, 40 }

' Outputs all of the element values (" 10 20 30 40")
For position As Integer = LBound(array) To UBound(array)
    Print array(position) ;
Next
```

Fixed-length and variable-length

There are two fundamental kinds of arrays: *fixed-length* and *variable-length*. The difference between the two is that the bounds of fixed-length arrays cannot be changed; they always store the same number of elements in the same positions. The bounds of variable-length arrays can be changed, affecting the number of elements stored and the positions of those elements.

Since fixed-length arrays never change size, the compiler chooses to store the memory for the array elements either in static storage or on the heap. The location of the array's **storage class**. This can be an advantage, since the cost of changing the size of an array doesn't include any adverse run-time penalty. Fixed-length arrays are declared using `Extern`, `Static` and `Dim`. At least an upper bound must be specified, and it must be a compile-time constant value, such as numeric literals, `Const` variable, or a constant expression.

Variable-length arrays can change in size, so the compiler chooses to store array elements at run-time, in the free store. The advantage here of dynamically resizing the arrays, however, run-time performance could be reduced if the array is resized or destroyed. Variable-length arrays are declared using `Extern`. When using `Extern`, `Static` or `Dim`, the lower and upper bounds can be specified in an empty array--or either one must have a variable value, such as a `Function` result. `ReDim` can be used to resize an existing variable-length array, and `ReDim Preserve` can be used to resize an existing variable-length array while preserving its lower and/or upper bounds.

```
' Creates a fixed-length array that holds 5 single
Const totalSingles = 5
Dim flarray(1 To totalSingles) As Single

' Creates an empty variable-length array that holds
Dim vlarray() As Integer

' Resizes the array to 10 elements.
ReDim vlarray(1 To 10) As Integer
```

Multi-dimensional arrays

The arrays discussed so far have been one-dimensional, that is, the elements are accessed through a single position. One-dimensional arrays can be thought of as a single line of elements. Arrays can also have more than one dimension; an individual element is accessed using two or more positions. Two-dimensional arrays use two positions--to refer to individual elements, like a grid or table. Three-dimensional arrays use three positions--a row, column and perhaps depth position--to refer to individual elements. Four-dimensional arrays can be thought of as one or more three-dimensional arrays. Multi-dimensional arrays are declared just like one-dimensional arrays, but the lower and upper bound range is specified.

```
' Take Care while initializing multi-dimensional arrays
Dim As Integer multidim(1 To 2, 1 To 5) = {{0,0,0,0,0}}
```

See also

- **Fixed-length Arrays**
- **Variable-length Arrays**
- **Variable Scope**

Fixed-length Arrays



Fixed-size homogeneous data structures.

Overview

Fixed-length arrays are **arrays** that have a fixed constant size through out the program. The memory used by a fixed-length array to store its elements is allocated on the stack or in the `.BSS` or `.DATA` sections of the executable, depending on how they are defined. This may allow for quicker program execution since the memory is pre-allocated, unlike **variable-length arrays**, whose element memory isn't allocated until the array is used.

Fixed-length arrays with **automatic storage**, have their elements allocated on the stack. Pointers to these elements remain valid only while the array is in scope. Fixed-length arrays with **static storage** are allocated in the `.DATA` or `.BSS` sections of the executable. Whether or not they are initialized when defined, so pointers to these elements remain valid for the entire execution of the program. Fixed-length arrays of any storage class can be used in C. However, for program execution, only **variable-length arrays** can be used.

Fixed-length arrays may also be used as data members inside **user-defined types**. The array is directly allocated as part of the user-defined type structure.

Declaration

A fixed-length array is declared with either the `Dim` or `Static` keywords followed by a parenthesized list of boundaries and an element **data type**.

```
' ' Defines a one-dimensional fixed-length array of type INTEGER having automatic storage.
Dim arrayOfIntegers(69) As Integer

' ' Defines a one-dimensional fixed-length array of type SHORT having static storage.
Static arrayOfShorts(420) As Short
```

There are various ways to specify an array's amount of elements. Each

dimensions. Each dimension has a lower bound and an upper bound.

```
Dim a(1) As Integer ' 1-dimensional, 2 elements
Dim b(0 To 1) As Integer ' 1-dimensional, 2 elements
Dim c(5 To 10) As Integer ' 1-
dimensional, 5 elements (5, 6, 7, 8, 9 and 10)

Dim d(1 To 2, 1 To 2) As Integer ' 2-
dimensional, 4 elements: (1,1), (1,2), (2,1), (2,2)
Dim e(255, 255, 255, 255) As Integer ' 4-
dimensional, 256 * 256 * 256 * 256 elements
```

For an array to be declared fixed-length, the boundaries must be specified as `Const` values or `Enum` constants.

```
Const myLowerBound = -5
Const myUpperBound = 10

' Declares a one-dimensional fixed-
length array, holding myUpperBound - myLowerBound + 1 elements
Dim arrayOfStrings(myLowerBound To myUpperBound) As String

' Declares a one-dimensional fixed-length array of
' big enough to hold an INTEGER.
Dim arrayOfBytes(0 To SizeOf(Integer) - 1) As Byte
```

Variable-length Arrays



Resizable homogeneous data structures. Also known as "dynamic array"

Overview

Variable-length arrays are **arrays** that can, during program execution, have their dimension[s] use a different subscript range. The memory is allocated at runtime in the heap, as opposed to fixed-length arrays which are in the .BSS or .DATA sections of the executable, depending on whether they are global or local.

Variable-length arrays may also be used as data members inside **user-defined types**, though, the array is not allocated as part of the user-defined type structure. Instead, the user-defined type only contains the array descriptor, and the array is still allocated on the heap, as if it were a global variable.

Variable-length arrays are often called "dynamic arrays" because their size is not being fixed-size.

Declaration

A variable-length array is declared with either the **Dim** or **ReDim** keyword, followed by the array name, a list of boundaries and an element **data type**. For an array to be declared with variable (non-constant) boundaries, **ReDim** must be used. **Dim** always has constant boundaries.

```
' ' Declares a one-dimensional variable-length array of integers, with initially 2 elements
ReDim a(0 To 1) As Integer

' ' Declares a 1-dimensional variable-length array
' ' It must be resized using Redim before it can be used
Dim b(Any) As Integer

' ' Same, but 2-dimensional
Dim c(Any, Any) As Integer

Dim myLowerBound As Integer = -5
```

```

Dim myUpperBound As Integer = 10

' Declares a 1-dimensional variable-length array
constant) boundaries.
' The array will have myUpperBound - myLowerBound
Dim d(myLowerBound To myUpperBound) As Integer

' Declares a variable-length array whose amount c
' by the first Redim or array access found. The a
' be resized using Redim before it can be used fo
Dim e() As Integer

```

Resizing

Resizing a variable-length array refers to "redefining" the array with di Elements outside the new subscript range[s] are erased; object eleme size, new elements are added initialized with a zero or *null* value; obje arrays are resized using the **ReDim** keyword following the same form a omitted from the **ReDim** statement.

```

' Define an empty 1-dimensional variable-length a
Dim array(Any) As Single

' Resize the array to hold 10 SINGLE elements...
ReDim array(0 To 9) As Single

' The data type may be omitted when resizing:
ReDim array(10 To 19)

```

Resizing an array cannot change its amount of dimensions, but only th

By default, element values of a variable-length array are lost when res resize, use the **Preserve** keyword.

Array Index



An array index is the number used to access an **Array** of **Variables** created.

Description

The following examples illustrate the use of array elements.

If we have an array `myArray` with elements of 1 to 10, filled with random

Index	Data
1	5
2	2
3	6
4	5
5	9
6	1
7	0
8	4
9	5
10	7

One can access each piece of data separately by pointing to the Index.

```
Print myArray(5)
```

Printing the data contained in the fifth element of `myArray` results in an

```
9
```

To change the contents of an array, use it like any other **Variable**:

```
myArray(3) = 0
```

To print the contents of `myArray(3)`, use the command:

```
Print myArray(3)
```

Which results in an output of:

```
0
```

Array elements can be indexed using another **Variable**. In this example

```
Dim a As Integer
For a = 1 To 10
    myArray(a) = 0
Next a
```

To change a random array element to a random value:

```
Dim Index As Integer
Dim Value As Integer
index = Int(Rnd(1) * 10) + 1 'This line will s
Value = Int(Rnd(1) * 10) + 1 'This line will c
myArray(index) = Value
```

Example

```
Declare Sub PrintArray()

Dim Numbers(1 To 10) As Integer
Dim Shared OtherNumbers(1 To 10) As Integer
Dim a As Integer

Numbers(1) = 1
```

```
Numbers(2) = 2
OtherNumbers(1) = 3
OtherNumbers(2) = 4

PrintArray ()

For a = 1 To 10
    Print Numbers(a)
Next a

Print OtherNumbers(1)
Print OtherNumbers(2)
Print OtherNumbers(3)
Print OtherNumbers(4)
Print OtherNumbers(5)
Print OtherNumbers(6)
Print OtherNumbers(7)
Print OtherNumbers(8)
Print OtherNumbers(9)
Print OtherNumbers(10)

Sub PrintArray ()
    Dim a As Integer
    For a = 1 To 10
        Print otherNumbers(a)
    Next a
End Sub
```

See also

- **Arrays**
- **Dim**
- **Function**
- **Sub**
- **Variables**
- **Variable Scope**

Pointers



Data types whose values are addresses in memory.

Declaration

Pointers are **Variables** whose values are addresses in memory, and the type of data that is pointed to depends on the type of pointer (e.g., pointer to **Integer** data). Pointers are declared like any other variable, with "ptr" following the type name.

Accessing pointed to data

The data pointed to by a pointer can be accessed with **operator** * (Value returned is a reference to the data that its operand points to). The following

```
Dim myInteger As Integer = 10
Dim myPointer As Integer Pointer = @myInteger
*myPointer = 20
Print myInteger
```

defines an **Integer** variable called myInteger and an **Integer** pointer called myPointer. The value of 10 is assigned to the location in memory where myInteger is stored. **operator** @ (Address of) returns the address of myInteger. The value of 20 is assigned to the location at the address of myInteger, or @myInteger. Changes to *myPointer directly affect the value of myInteger (the expression "*myPointer" is the same thing as "myInteger").

Pointers to user-defined types

Pointers to user-defined types are defined and used like all other pointers. The declaration of a **Type** or **Class** requires one of the following two methods:

```
Type myType
    a As Integer
    b As Double
End Type
```

```

Dim x As myType
Dim p As myType Pointer = @x

'' 1) dereference the pointer and use the member a
(*p).a = 10
(*p).b = 12.34

'' 2) use the shorthand form of the member access
Print p->a
Print p->b

```

The first method uses **Operator . (Member Access)**. This operator accesses references, so the pointer is dereferenced first. The member access comes over the dereference operator, so parentheses are needed to dereference it with the member access operator.

The second method uses **Operator -> (Pointer To Member Access)**. It accesses members from pointers, which are automatically dereferenced. This is clearer, although both forms produce identical results.

See also

- **Operator @ (Address Of)**
- **Operator * (Value Of)**
- **Operator . (Member Access)**
- **Operator -> (Pointer To Member Access)**
- **VarPtr**
- **StrPtr**
- **ProcPtr**

Pointer Arithmetic



Manipulating address values mathematically.

Overview

Adding and subtracting from pointers
Incrementing and decrementing pointers
Distance between two pointers

Overview

It is often useful to iterate through memory, from one address to another. Pointers are used to accomplish this. While the type of a pointer determines the type of variable or object retrieved when the pointer is dereferenced (`Operator * (Value of)`), it also determines the *distance*, in bytes, its pointer type takes up in memory. For example, a **Short** takes up two (2) bytes of memory, while a **Single** needs four (4) bytes.

Adding and subtracting from pointers

Pointers can be added to and subtracted from just like a numeric type. The result of this addition or subtraction is an address, and the type of pointer determines the distance from the original pointer.

For example, the following,

```
Dim p As Integer Ptr = New Integer[2]

*p = 1
*(p + 1) = 2
```

will assign the values "1" and "2" to each integer in the array pointer to which `p` points. Since `p` is an **Integer Pointer**, the expression "`*(p + 1)`" is saying to dereference an **Integer** four (4) bytes from `p`; the "1" indicates a distance of "the size of an **Integer**", or four (4) bytes.

Subtraction follows the exact same principle. Remember, $a - b = a + -$.

Incrementing and decrementing pointers

Sometimes it is more convenient to modify the pointer itself, in which combination addition and subtraction operators will work just like above example, the following,

```
Dim array(5) As Short = { 32, 43, 66, 348, 112, 0 }
Dim p As Short Ptr = @array(0)

While (*p <> 0)
    If (*p = 66) Then Print "found 66"
    p += 1
Wend
```

iterates through an array until it finds an element with the value of "0". an element with the value "66" it displays a nice message.

Distance between two pointers

The distance between two pointers is retrieved with **operator -** (Subt) and is measured in values, not bytes. For example, the following,

```
Type T As Single

Dim array(5) As T = { 32, 43, 66, 348, 112, 0 }
Dim p As T Ptr = @array(0)

While (*p <> 0)
    p += 1
Wend
Print p - @array(0)
```

will output "5" regardless of what type τ is. This is because there is a fixed element difference between the first element of *array* (32) and the element pointed to by p (0).

Specifically, if a and b are both pointers of type T , the distance between them is the number of bytes between them, divided by the size, in bytes, of

`Abs(cast(byte ptr, a) - cast(byte ptr, b)) / SizeOf(T)`

See also

- **Operator + (Add)**
- **Operator - (Subtract)**
- **Operator @ (Address Of)**
- **Operator * (Value Of)**
- **Pointer Operators**

Implicit Declarations



Lazy declaration of variables.

The *qb* and *fblite* FreeBASIC language dialects allow variable names to be used without declaring them first. This is called implicit or lazy declaration since the actual declaration is inferred from how the name is first used.

Variable Type

When a variable is implicitly declared, its type depends on one of two things: the most recent default implicit type directive, if any, or the variable type suffix symbol used, if any.

Default type

In the *qb* dialect, implicitly declared variables default to **Single** type, while in the *fblite* dialect they default to **Integer** type.

Default implicit type directives

"DEFxxx" directives dictate the new default type for any following implicit variable declarations. These directives are: **DefByte**, **DefUByte**, **DefShort**, **DefUShort**, **DefInt**, **DefUInt**, **DefLng**, **DefSng**, **DefDb1** and **DefStr**.

Variable type suffix symbols

Variable names suffixed with one of a certain set of symbols will be implicitly declared of a certain type. These symbols are: '%' for **Integer**, '&' for **Long**, '!' for **Single**, '#' for **Double** and '\$' for **String**. These symbols override previous "DEFxxx" directives, if any.

Implicit Array Declaration

Currently, FreeBASIC does not support implicit declaration of arrays.

Debugging

For full debugging support, all variables must be explicitly declared and suffixes should not be used. The use of **Option Explicit** is recommended to turn off support for implicit declarations, so that mistyped variable names are caught at compile time by the compiler.

See also

- **Option Explicit**
- **FreeBASIC Language Dialects**

Variable Initializers



Variable initializers are supported for initializing Arrays, variables and UDTs.

Syntax

```
Dim scalar_symbol [AS DataType] = expression
Dim array_symbol ([lbound TO] ubound) [AS DataType] => { expression }
Dim udt_symbol AS DataType = ( expression [, ...] )
```

Description

Arrays, variables and UDTs may be given a value at the time of their declaration, as shown above. Please note the important differences between initializing variables and arrays. Variables are initialized as they would in a normal assignment, using an equals sign (=). Arrays are initialized using an equals sign followed by a greater than symbol (=>). Array values are given in curly brackets, and UDT values are given in comma delimited lists.

These methods of initializing variables can be nested within one another to initialize a multidimensional array:

```
Dim array(1 To 2, 1 To 5) As Integer => {{1, 2, 3,
```

In this declaration, the values for the left-most dimension are given as arrays of any dimension to be initialized.

UDTs and arrays can be nested within each other as well. For instance, the following code initializes an array of UDTs.

```
Type mytype
    var1 As Double
    var2 As Integer
    var3 As ZString Ptr
End Type

Dim MyVar(2) As mytype => _
    { _
```

```
    (1.0, 1, @"Hello"), _  
    (2.0, 2, @"GoodBye") _  
}
```

For module-level, static, or global variables, initialized values must be report a compile-time error if otherwise.

Differences from QB

- Variable Initializers are new to FreeBASIC

See also

- [Dim](#)

Visibility and lifetime of variables, objects and arrays

A variable, object or array's storage class determines when and where memory is allocated for it and when that memory is destroyed. There are 2 storage classes in FreeBASIC: *automatic* and *static*.

Automatic

Automatic variable, object and array lifetimes begin at the point of declaration and end when leaving the scope they are declared in.

Automatic entities are guaranteed to have unique storage for each instance of the block in which they are declared. For example, the *automatic* variables declared within a procedure will be allocated at different addresses and have unique state (value) for each call to the procedure.

Automatic variables, objects and arrays are defined using the **Dim**, **ReDim** and **Var** keywords without the **Shared** specifier.

The memory for *automatic* variables, objects and arrays is allocated on the program stack.

Automatic variables, objects and arrays have no linkage.

Static

Static variable, object and array lifetimes begin at program creation and end with program termination.

Static entities are guaranteed to have the same storage for each instance of the block in which they are declared. For example, the *static* variables declared within a procedure will be allocated at the same address, and retain their state (value) across each call to the procedure.

Static variables, objects and arrays are declared using the **Static** keyword. Entities declared using the **Shared** specifier are implicitly *static*. All entities declared within a procedure that is declared using the **Static** specifier are also implicitly *static*.

The memory for *static* variables, objects and arrays is allocated in the `.BSS` section of the executable, or in the `.DATA` section if they are initialized when defined. *Static* variable-length arrays must be declared empty, with an empty subscript range list; their element data is still allocated in the free store (when they are resized), but the internal array data is allocated in the `.DATA` section of the executable to allow the element data to persist throughout program execution.

Static variables, objects and arrays have internal linkage by default, unless previously declared using the **Extern** or **Common** keywords.

Platform Differences

- In DOS and Windows platforms, the size of the program stack can be adjusted at compile-time using the `-t` **command-line switch**. In Linux platforms, the size of the program stack can be adjusted at load-time by modifying `/etc/security/limits.conf`, or on a per-thread basis using the shell builtin `ulimit`.

Differences from QB

- QuickBASIC allows *static* entities to be declared within procedures and `DEF FN` routines only.

See also

- **Extern, Common**
- **Dim, ReDim, Var, Shared**
- **Static**
- **Linkage**

Variable Scope



Visibility and access rules for variables and objects

A variable's scope refers to its visibility in a program. A variable is not visible where it was declared. Where and how a variable is declared determines its scope.

In FreeBASIC, there are 4 categories of scope: **local**, **shared**, **common**, and **global**, each with different visibility rules, which are detailed below.

Local Scope

Variables declared in the local scope are visible only in the most local block in which they are declared.

- **Sub**, **Function**, the main body, and each compound statement in a block
- Explicitly declared variables using **Dim** or **ReDim** take the scope of the block in which they are declared
- Implicit variables take the scope of the the local most **Scope...** block in which they are declared

In the local scope, there is no visibility between module-level code and code within a block. A decision or loop statement will only be visible within the block in which it is declared. Variables declared in the local scope of a module are not visible in any of the functions within the module, and variables declared in functions are not visible in the module-level code, nor any other functions.

Variables declared inside **Scope** blocks may only be declared of local scope, but they will inherit the surrounding scope, so local variables declared outside a block (in the *program*).

You can declare a variable to be of local scope explicitly by using the **Local** keyword (see **Implicit Declarations**). The example program `local.bas` demonstrates this.

local.bas

```
' ' visible only in this module
Dim As Integer local_moduleLevel1
```

```

'' OK.
Print local_moduleLevel1

Scope
'' OK; SCOPE Blocks inherit outer scope
Print local_moduleLevel1

'' visible only in this SCOPE Block
Dim As Integer local_moduleLevel2

'' OK.
Print local_moduleLevel2
End Scope

'' Error; can't see inner-SCOPE vars
'' print local_moduleLevel2

Function some_function( ) As Integer
'' visible only in this function
Dim As Integer local_functionLevel

'' OK.
Print local_functionLevel

'' Error; can't see local module-level vars
'' print local_moduleLevel1

'' Error; can't see local module-level vars
'' print local_moduleLevel2

Function = 0

End Function

'' print local_functionLevel
End 0
''

```

Shared Scope

Variables declared in the shared scope of a module are visible to both

Unlike the local scope, the shared scope makes module-level variable module *shares* its declarations with its functions.

Variables can only be declared to be of shared scope at the module-level functions nor **Scope** blocks can declare variables in the shared scope, function or block.

You can declare a variable to be of shared scope by using the DIM statement `shared_scope.bas` demonstrates visibility rules for the shared scope.

shared.bas

```
' ' visible throughout this module
Dim Shared As Integer shared_moduleLevel1

' ' OK.
Print shared_moduleLevel1

Scope
    ' ' OK; can see outer-scope vars
    Print shared_moduleLevel1

    ' ' Error; SCOPE-level vars cannot be shared
    ' ' dim shared as integer shared_ModuleLevel2
End Scope

End 0

Function some_function( ) As Integer
    ' ' OK; can see shared module-level vars
    Print shared_moduleLevel1

    ' ' Error; function-level vars cannot be shared
    ' ' dim shared as integer sharedFunctionLevel
```

```
Function = 0  
End Function
```

Common Scope

Variables declared in the common scope are visible to all modules.

Variables declared with **Common** are visible to other modules with a mat declared must match from between modules.

module1.bas

```
' ' compile with:  
' ' fbc -lang qb module1.bas module2.bas  
  
'$lang: "qb"  
  
Declare Sub Print_Values()  
Common m1 As Integer  
Common m2 As Integer  
  
m1 = 1 ' This is executed after all  
  
Print "Module1"  
Print "m1 = "; m1 ' m1 = 1 as set in this modu  
Print "m2 = "; m2 ' m2 = 2 as set in module2  
  
Print_Values
```

module2.bas

```
Common m1 As Integer  
Common m2 As Integer  
  
m2 = 2
```

```

Print "Module2"           ' This is executed first
Print "m1 = "; m1         ' m1 = 0 (by default)
Print "m2 = "; m2         ' m2 = 2

Sub Print_Values()
    Print "Module2.Print_Values"
    Print "m1 = "; m1     ' Implicit variable = 0
    Print "m2 = "; m2     ' Implicit variable = 0
End Sub

```

Output:

```

Module2
m1 = 0
m2 = 2
Module1
m1 = 1
m2 = 2
Module2.Print_Values
m1 = 0
m2 = 0

```

Common Shared Scope

Variables declared in the common shared scope are visible to all mod

Variables declared with **common** are visible to other modules with a mat declared must match from between modules. Within a module the **sha** and makes the variable visible to all subs and functions.

module3.bas

```

'' compile with:
''     fbc module3.bas module4.bas

Declare Sub Print_Values()

```

```

Common m1 As Integer
Common m2 As Integer

'' This is executed after all other modules
m1 = 1

Print "Module3"
Print "m1 = "; m1      '' m1 = 1 as set in this mod
Print "m2 = "; m2      '' m2 = 2 as set in module2

Print_Values

```

module4.bas

```

Common Shared m1 As Integer
Common Shared m2 As Integer

m2 = 2

Print "Module4"      '' This is executed first
Print "m1 = "; m1    '' m1 = 0 (by default)
Print "m2 = "; m2    '' m2 = 2

Sub Print_Values()
    Print "Module4.Print_Values"
    Print "m1 = "; m1    '' m1 = 1
    Print "m2 = "; m2    '' m2 = 2
End Sub

```

Output:

```

Module4
m1 = 0
m2 = 2
Module3
m1 = 1
m2 = 2

```

```
Module4.Print_Values  
m1 = 1  
m2 = 2
```

Example

See examples above.

See also

- **Scope**
- **Dim**
- **Common**
- **Shared**
- **Variables**
- **Implicit Declarations**

Variable and Procedure Linkage



Name visibility within and between modules

Linkage refers to the visibility of the name of a variable, object or procedure between one or more modules of a program. In other words, linkage dictates how a name is shared between modules. There are two main types of linkage a name can have: *internal* and *external*.

Internal linkage

Names with *internal linkage* only refer to variables, objects or procedures defined within their own module; they are not outwardly visible to other modules. This means that two or more modules can refer to different things using the same name. Note that linkage only refers to visibility of a name, and depending on storage class and lifetime, a variable, object or procedure with internal linkage may be shared between modules using its address.

Module-scope declarations

Variable and object names declared at module-scope have internal linkage unless otherwise declared with **Extern** or **Common**. For example variable names first introduced with **Dim** or **Static** have internal linkage and those variables can only be referred to by name within the module in which they are defined. Note that using **Shared** only allows name visibility within the module's procedures, and does not contribute to the name's linkage.

Procedure names declared with **Private** have internal linkage.

Local-scope declarations

All variable and object names declared at local-scope (in a **Do** loop, or procedure body, for instance) have internal linkage.

External linkage

Names with *external linkage* may refer to variables, objects or procedures defined within their module or in another module. Having external linkage means that a name is outwardly visible to other modules, and all modules that use that same external name all refer to the same variable, object or procedure. Thus, only one module may define an external name (the compiler will complain about a duplicate definition if it finds an additional definition of a name with external linkage).

Module-scope declarations

Variable and object names declared at module-scope are declared to have external linkage with **Extern** or **Common**.

Extern declares the variable having external linkage, but does not define it. This external declaration must come before any definition of the same name (a declaration without **Extern** specifies internal linkage and currently, any further external declarations of that name signify a duplicated definition). Variable and object names with external linkage declared using **Extern** are always in the shared scope, and so can be referred to within procedure bodies.

Common declares the variable having external linkage as well as defining the variable. But, it is different from **Extern** in that the **Common** definition of the variable may appear in more than one module. When used with arrays, only variable-length arrays without subscripts may be declared and the array must be sized at run-time using **Dim** or **ReDim** before it can be used. Variable and object names with external linkage declared using **Common** are only in the shared scope if the **Shared** scope specifier is also given. Shared variables can be referred to within procedure bodies.

When both **Extern** and **Common** are both used to declare and define a variable, the effect is that the meaning of **Common** statement is altered to behave as though it were a **Dim** declaration. So it is generally, not

recommended to mix **Extern** and **Common** on the same variable in the same module. However, variables may be declared and defined with **Common** in one module and then referenced with **Extern** in another module without confusion.

Procedure names are declared to have external linkage by default. Declarations using **Public** explicitly specify external linkage.

Local-scope declarations

Currently, names declared at local-scope cannot have external linkage.

User Defined Types



Custom types.

Overview

User-Defined Types are special kinds of **variables** which can be created. A User-Defined Type (UDT) is really just a container that contains a bunch of variables. Unlike arrays UDTs can hold *different* variable types (whereas arrays hold the *same* type). In fact, UDTs can even have **procedures** inside of them!

Members

The different variables and/or procedures stored inside a UDT are called members. Members can be variables of just about any type, including numbers, **Enums**, and even arrays. Variables are created in UDTs much the same way as normally, except that the Dim keyword is optional. UDT members are accessed by the name of the UDT variable followed by ".someVar". Here is an example:

```
'Define a UDT called myType, with an Integer member
Type myType
  As Integer someVar
End Type

'Create a variable of that type
Dim myUDT As myType

'Set the member someVar to 23, then display its contents
myUDT.someVar = 23
Print myUDT.someVar
```

Notice that the **Type...End Type** does not actually create a variable of that type. You must create a variable of that type to use it.

UDT Pointers

UDT Pointers are, as the name implies, pointers to UDTs. They are created and there is a special way to use them. To access the member of a UDT pointer `p` -> **Operator**. For example, if `myUDTPtr` is a pointer to a UDT which has a member `someVar`, you can access the member as `myUDTPtr->someVar`, which is a much cleaner syntax than `(myUDTPtr).someVar`.

```
Type rect
    x As Integer
    y As Integer
End Type

Dim r As rect
Dim rp As rect Pointer = @r

rp->x = 4
rp->y = 2

Print "x = " & rp->x & ", y = " & rp->y
Sleep
```

See also

- [Type Aliases](#)
- [Temporary Types](#)
- [Constructors and Destructors](#)
- [Member Procedures](#)
- [Member Access Rights](#)
- [Operator Overloading](#)

Type Aliases



Additional names for variable or object types

Overview

Declaration

Overload resolution

Pointers to procedure pointers

Type forwarding

Incomplete types

Overview

Type aliases are alternative names for a type. They can be used to fa one type to another, save typing, or make circular dependency possib

Declaration

Type aliases are declared using the **Type** keyword much like declaring **Extern** or **Dim**.

The following example declares a type alias to **Single** called "*float*", & initializes two variables of that type:

```
Type float As Single

Declare Function add (a As float, b As float) As f

Dim foo As float = 1.23
Dim bar As float = -4.56
```

Procedure pointer type aliases are declared in the same fashion, as s

```
Declare Function f (ByRef As String) As Integer
```

```

Type func_t As Function (ByRef As String) As Integer

Dim func As func_t = @f

Function f (ByRef arg As String) As Integer
    Function = CInt(arg)
End Function

```

Overload resolution

Type aliases are just that - aliases. For all intents and purposes, a type as far as procedure overload resolution is concerned, a procedure declared with a parameter of type *"alias_to_T"* is the same as a procedure declared with a parameter of type *T* (as well as overloading member procedures as well).

In other words, it is an error - duplicated definition - to declare a procedure in a type and its alias, as the following example shows:

```

Type float As Single

Declare Sub f Overload (a As Single)

' ' If uncommented, this will generate a duplicated definition
' ' Declare Sub f (a As float)

```

Pointers to procedure pointers

Pointers to procedure pointers are just like any other pointer type, except they are pointers to procedure pointers. Because the syntax for declaring procedure pointers doesn't allow a pointer to procedure pointer when the procedure is a function (because of the way the language handles procedure pointers), a type alias is used.

The following example declares a pointer to a procedure returning an integer, and then a pointer to a pointer to a procedure returning an integer:

```
Dim pf As Function() As Integer Ptr

Type pf_t As Function() As Integer
Dim ppf As pf_t Ptr
```

Type forwarding

Type aliases can be forward referencing: an alias can refer to some o

```
Type foo As bar

Type sometype
  f As foo Ptr
End Type

Type bar
  st As sometype
  a As Integer
End Type
```

Using a type alias and forward referencing allows circular dependenci

```
Type list As list_

Type listnode
  parent As list Ptr
  text As String
End Type

Type list_
```

```

first As listnode Ptr
count As Integer
End Type

```

Incomplete types

A type is considered incomplete until the size of it, that is the number of bytes it occupies in memory is known, and the offsets of all of its fields are known. It is not possible to declare a variable of an incomplete type, pass an incomplete type as a parameter, or access the fields of an incomplete type.

However, pointers to incomplete types may be allocated, declared as variables, and passed as parameters to a procedures since the size of a pointer is known.

```

Type sometype As sometype_

'' Not allowed since size of sometype is unknown
'' TYPE incomplete
''   a AS sometype
'' END TYPE

'' Allowed since size of a pointer is known
Type complete
  a As sometype Ptr
End Type
Dim x As complete

'' Not allowed since size of sometype is still unknown
'' DIM size_sometype AS INTEGER = SIZEOF( sometype )

'' Complete the type
Type sometype_
  value As Integer
End Type

```

```
' ' Allowed since the types are now completed
Dim size_sometype As Integer = SizeOf( sometype )

Type completed
  a As sometype
End Type

Dim size_completed As Integer = SizeOf( completed
```

Constructors and Destructors



In charge of the creation and destruction of objects.

Overview

Declaration

Default constructors

Copy constructors

Calling constructors

Overview

Constructors and destructors are responsible for creating and destroy constructors give objects their initial state, that is, they give meaningful data. Destructors perform the opposite function; they make sure any r properly freed.

Simply, constructors are special member procedures that are called w destructors are special member procedures called when an object is c destructors are called automatically by the compiler whenever an obje explicitly with the use of the **Dim** or **New** keywords, or implicitly by passi value or through an object going out of scope.

Declaration

Constructors and destructors are declared like member procedures b instead of **Sub** or **Function**, and without a name. Similarly, they are def or **Class** they are declared in.

A **Type** or **Class** can have multiple constructors, but only one destructo

Default constructors

Default constructors are constructors that either have no parameters, default value. They are called when an object is defined but not initiali array, with the **Dim**, **ReDim** or **New[]** keywords. The first constructor decla default constructor.

Copy constructors

Copy constructors are constructors called when an object is created, (same type (or an object that can be converted to that type). This happens when an object is created with another object, or implicitly by passing an object to a procedure. Copy constructors are declared having one parameter: an object of the same type passed by reference.

Copy constructors are only called when creating and initializing object handled by the `Member Operator Let`.

Calling constructors

Unlike other member procedures, constructors are generally not called. Instead, a constructor is specified in a `Dim` statement either with an initializer statement with or without arguments.

When specifying an initializer for an object, the name of the type followed by the number of objects to be created is used.

```
Type foo
    ' ' Declare a default ctor, copy ctor and normal ctor
    Declare Constructor
    Declare Constructor (ByRef As foo)
    Declare Constructor (As Integer)

    ' ' Declare a destructor
    Declare Destructor

    ints As Integer Ptr
    numints As Integer
End Type

' ' Define a constructor that creates 100 integers
Constructor foo
    ints = New Integer(100)
    numints = 100
End Constructor
```

```

'' Define a constructor that copies the integers f
Constructor foo (ByRef x As foo)
    ints = New Integer(x.numints)
    numints = x.numints
End Constructor

'' Define a constructor that creates some integers
Constructor foo (n As Integer)
    ints = New Integer(n)
    numints = n
End Constructor

'' Define a destructor that destroys those integer
Destructor foo
    Delete[] ints
End Destructor

Scope
    '' calls foo's default ctor
    Dim a As foo
    Dim x As foo Ptr = New foo

    '' calls foo's copy ctor
    Dim b As foo = a
    Dim y As foo Ptr = New foo(*x)

    '' calls foo's normal ctor
    Dim c As foo = foo(20)
    Dim z As foo Ptr = New foo(20)

    '' calls foo's dtor
    Delete x
    Delete y
    Delete z
End Scope '' <- a, b and c are destroyed here as w

```


Member Procedures



Procedures with full access to members of a **Type** or **Class**.

Declaration and definition

Declaring and defining member procedures.

Usage

Calling member procedures.

The hidden parameter, This

Implicit access to the instance with which non-static member procedures

Access rights

Referring to other members in member procedures.

Overloading

Declaring two or more member procedures with the same name.

Static member procedures

Differences from non-static member procedures.

The term 'member procedure' refers to both static and non-static member

Declaration and definition

Member procedures are declared much like normal module-level procedures within, and defined outside, a **Type** or **Class** definition [1].

When defining member procedures, the procedure name is prefixed with the member access operator (**Operator . (Member Access)**). It is an error without a matching declaration in the **Type** or **Class** definition.

The following example declares and defines a **Sub** and **Function** member

```
' ' foo1.bi

Type foo
  Declare Sub f (As Integer)
  Declare Function g As Integer

  i As Integer
End Type
```

```

Sub foo.f (n As Integer)
    Print n
End Sub

Function foo.g As Integer
    Return 420
End Function

```

Usage

Member procedures are referred to just like member data, that is, their object instance and the member access operator (**Operator** . (**Member**

The following example, using the code from the last example, calls **su**

```

'' ... foo with non-static members as before ...
#include once "foo1.bi"

Dim bar As foo
bar.f(bar.g())

```

The hidden parameter, This

Member procedures actually have an additional parameter than what are called, using the name of an instance and **Operator** . (**Member Ac** passed along with any other arguments in the call, allowing the memb instance.

The additional parameter added by the compiler is called **This**, and si **This** are actually modifications to the instance that was passed to the You can use **This** just like any other variable, ie., pass it to procedures other member procedures and access member data using **Operator** .

Most of the time, however, using **This** explicitly is unnecessary; members of the instance which they are passed directly by name, with **Operator . (Member Access)**. The only times when you need to qualify member name is hidden, for example, by a parameter or local variable member name is the only way to refer to these hidden member name:

Note:

*To access duplicated symbols defined outside the Type, use: .Sort inside a **With..End With** block).*

The following example uses the **This** keyword to refer to member data and local variable:

```
Type foo
  Declare Sub f (i As Integer)
  Declare Sub g ()

  i As Integer = 420
End Type

Sub foo.f (i As Integer)
  ' A parameter hides T.i, so it needs to be qualified
  Print this.i
End Sub

Sub foo.g ()
  ' A local variable hides T.i, so it needs to be qualified
  Dim i As Integer
  Print this.i
End Sub
```

Access rights

Unlike normal module-level procedures, member procedures have full **Type** or **class** they are declared in; they can refer to the public, protected **Class**.

Overloading

A member procedure can be declared to have the same name as another member procedure with the same name as long as the parameters are different, either in number or in type. This is referred to as overloading.

Only the parameters are used to determine if a procedure declaration or **class** could have static and non-static member procedures with the same name as other member procedures with the same name.

Unlike a module-level procedure, which needs to specify the **overload** keyword to indicate overloading, a member procedure is overloadable by default, and does not need the **overload** keyword.

```
Type T
  Declare Sub f

  '' Different number of parameters:
  Declare Sub f (As Integer)

  '' Different type of parameters:
  Declare Sub f (ByRef As String)

  '' Again, parameters are different:
  Declare Function f (As UByte) As Integer

  '' following three members would cause an error because the
  '' number of parameters and/or types do not differ from the
  '' previous members.
  '' Declare Function f As Integer
  '' Declare Function f (As UByte) As String
  '' Declare Static Function f (As UByte) As Integer

  '' ...
  somedata As Any Ptr
End Type
```

Static member procedures

Static member procedures are declared and defined much in the same way as non-static member procedures, with the **static** keyword preceding the declaration and definition.

Member procedures defined using the **static** keyword must be declared in the **Class** definition, or a compiler error will occur. Like non-static member procedures, a static member procedure without a matching declaration in the **Type** definition will cause a compiler error.

Do not confuse this with procedure definitions that specify static storage class by appending the **static** keyword to the procedure header. The **static** keyword is not applicable to member procedures; static member procedures can be defined with static variables.

The following example declares two static member procedures, the first for object storage. Note that the **static** keyword is optional in the member procedure definition.

```
' ' foo2.bi

Type foo
    Declare Static Sub f (As Integer)
    Declare Static Function g As Integer

    i As Integer
End Type

Static Sub foo.f (n As Integer) Static
    Print n
End Sub

Function foo.g As Integer
    Return 420
End Function
```

Static member procedures can be called like non-static member procedures. They can be called with the name of an instance and the member access operator.

They can also be called by qualifying the procedure name with the name of the class declared in and the member access operator (**Operator** . **Member Access**).

required in order to call static-member procedures.

The following example, using the code from the last example, uses both member and static member procedures:

```
' ' ... foo with static members as before ...  
#include once "foo2.bi"  
  
Dim bar As foo  
bar.f(foo.g())
```

Unlike non-static member procedures, which are declared with an external procedure declaration, static member procedures do not get passed an instance when called. Because of this, static member procedures can only refer to constants, enumerations, other static members (data or procedure names). Static member procedures can still refer to non-static member variables, for example: a parameter or local variable.

The following example refers to a non-static member from a static procedure:

```
Type foo  
    Declare Static Sub f (ByRef As foo)  
  
    i As Integer  
End Type  
  
Sub foo.f (ByRef self As foo)  
    ' Ok, self is an instance of foo:  
    Print self.i  
  
    ' would cause error  
    ' cannot access non-static members, no foo in scope  
    ' Print i  
End Sub
```

- [1] *In the future, member procedures may be able to be defined within*
- [2] *Static member procedures do not require an object instance in ord*
- [3] *Static member procedures do not have this extra parameter addec
the object instance from which it was called with.*

Properties



Properties are a special mix of member variable and member procedure set or retrieve values of an object, through normal looking assignments (also let the object perform actions if it needs to update itself).

Basic properties

Declaring and using setter and getter properties.

Indexed properties

Properties with an additional parameter.

Basic properties

A property is declared similar to a **member procedure**, except that `used` instead of `Sub` or `Function`. For example, let's consider a window system or GUI library.

```
Type Window
Private:
    As String title_
End Type

Dim As Window w
```

In order to set the window's title, a *setter* property can be added:

```
Type Window
    Declare Property title(ByRef s As String)
Private:
    As String title_
End Type

Property Window.title(ByRef s As String)
    this.title_ = s
```

```
End Property
```

```
Dim As Window w  
w.title = "My Window"
```

It is very similar to a member **Sub**, as it takes a parameter and updates state based on the parameter. However, the syntax for sending this parameter is an assignment, not a function call. By assigning the new value to the `title` property, the `Set Title` procedure will automatically be called with the given new value, and the window will reflect the change. It is up to the object how to represent the property.

By design, properties can only be assigned one value at a time, and a setter procedure can not have more than one parameter.

After setting the window title, it should also be possible to retrieve it. Here is the `Get Title` property:

```
Type Window  
    ' ' setter  
    Declare Property title(ByRef s As String)  
    ' ' getter  
    Declare Property title() As String  
Private:  
    As String title_  
End Type  
  
    ' ' setter  
Property Window.title(ByRef s As String)  
    this.title_ = s  
End Property  
  
    ' ' getter  
Property Window.title() As String  
    Return this.title_  
End Property
```

```
Dim As Window w
w.title = "My Window"
Print w.title
```

The getter is very similar to a **Function**. It is supposed to return the current value of the property, and it allows the current value to be calculated from other information. Note that both *setter* and *getter* use the same identifier, indicating they both operate on the same property.

Just like **method overloading**, it is possible to specify multiple setters with different parameter types:

```
Type Window
    Declare Property title(ByRef s As String)
    Declare Property title(ByVal i As Integer)
    Declare Property title() As String
Private:
    As String title_
End Type

Property Window.title(ByRef s As String)
    this.title_ = s
End Property

Property Window.title(ByVal i As Integer)
    this.title_ = "Number: " & i
End Property

Property Window.title() As String
    Return this.title_
End Property

Dim As Window w
w.title = "My Window"
Print w.title
w.title = 5
```

```
Print w.title
```

In comparison to this example of properties, here is similar code that c

```
Type Window
    Declare Sub set_title(ByRef s As String)
    Declare Sub set_title(ByVal i As Integer)
    Declare Function get_title() As String
Private:
    As String title
End Type

Sub Window.set_title(ByRef s As String)
    this.title = s
End Sub

Sub Window.set_title(ByVal i As Integer)
    this.title = "Number: " & i
End Sub

Function Window.get_title() As String
    Return this.title
End Function

Dim As Window w
w.set_title("My Window")
Print w.get_title()
w.set_title(5)
Print w.get_title()
```

The code is basically the same, only the syntax is different. Properties to combine the setter/getter concept and the language's normal way c **accessing** values to a class' member variables. It is up to the program they prefer.

Here is an example demonstrating a text user interface window class and title using properties:

```
Namespace tui
  Type Point
    Dim As Integer x, y
  End Type

  Type char
    Dim As UByte value
    Dim As UByte Color
  End Type

  Type Window
    ' public
    Declare Constructor _
      ( _
        x As Integer = 1, y As Integer = 1
        w As Integer = 20, h As Integer =
        title As ZString Ptr = 0 _
      )

    Declare Destructor

    Declare Sub show

    ' title property
    Declare Property title As String
    Declare Property title( new_title As Strin

    ' position properties
    Declare Property x As Integer
    Declare Property x( new_x As Integer )

    Declare Property y As Integer
    Declare Property y( new_y As Integer )
```

```

Private:
    Declare Sub redraw
    Declare Sub remove
    Declare Sub drawtitle

    Dim As String p_title
    Dim As Point Pos
    Dim As Point siz
End Type

Constructor Window _
(
    _
    x_ As Integer, y_ As Integer, _
    w_ As Integer, h_ As Integer, _
    title_ As ZString Ptr _
)

    pos.x = x_
    pos.y = y_
    siz.x = w_
    siz.y = h_

    If( title_ = 0 ) Then
        title_ = @"untitled"
    End If

    p_title = *title_
End Constructor

Destructor Window
    Color 7, 0
    Cls
End Destructor

Property window.title As String
    title = p_title
End Property

Property window.title( new_title As String )

```

```

        p_title = new_title
        drawtitle
    End Property

Property window.x As Integer
    Return pos.x
End Property

Property window.x( new_x As Integer )
    remove
    pos.x = new_x
    redraw
End Property

Property window.y As Integer
    Property = pos.y
End Property

Property window.y( new_y As Integer )
    remove
    pos.y = new_y
    redraw
End Property

Sub window.show
    redraw
End Sub

Sub window.drawtitle
    Locate pos.y, pos.x
    Color 15, 1
    Print Space( siz.x );
    Locate pos.y, pos.x + (siz.x \ 2) - (Len(
    Print p_title;
End Sub

Sub window.remove
    Color 0, 0
    Var sp = Space( siz.x )

```

```

        For i As Integer = pos.y To pos.y + siz.y
            Locate i, pos.x
            Print sp;
        Next
    End Sub

    Sub window.redraw
        drawtitle
        Color 8, 7
        Var sp = Space( siz.x )
        For i As Integer = pos.y + 1 To pos.y + si
            Locate i, pos.x
            Print sp;
        Next
    End Sub
End Namespace

Dim win As tui.window = tui.window( 3, 5, 50, 15 )

win.show
Sleep 500

win.title = "Window 1"
Sleep 250
win.x = win.x + 10
Sleep 250

win.title = "Window 2"
Sleep 250
win.y = win.y - 2
Sleep 250

Locate 25, 1
Color 7, 0
Print "Press any key...";

Sleep

```

Note how updating the window's position or title automatically causes

Indexed properties

Properties can have an additional parameter that is called an `index` (a `compare` parameter is allowed). The index is specified in parentheses behind the property value. If the property was an array (with only one dimension). For example:

```
Type IntArray
    ' ' setters
    Declare Property value(index As Integer, v As Integer) As Integer
    Declare Property value(index As String, v As Integer) As Integer
    Declare Property value(index As Integer, v As String) As String
    Declare Property value(index As String, v As String) As String

    ' ' getters
    Declare Property value(index As Integer) As Integer
    Declare Property value(index As String) As String

Private:
    Dim As Integer data_(0 To 9)
End Type

Property IntArray.value(index As Integer) As Integer
    Return This.data_(index)
End Property

Property IntArray.value(index As String) As Integer
    Return This.data_(CInt(index))
End Property

Property IntArray.value(index As Integer, v As Integer) As Integer
    This.data_(index) = v
End Property

Property IntArray.value(index As String, v As Integer) As Integer
```

```

        This.data_(CInt(index)) = v
    End Property

    Property IntArray.value(index As Integer, v As Str
        This.data_(index) = CInt(v)
    End Property

    Property IntArray.value(index As String, v As Stri
        This.data_(CInt(index)) = CInt(v)
    End Property

    Dim a As IntArray

    a.value(0) = 1234
    a.value("1") = 5678
    a.value(2) = "-1234"
    a.value("3") = "-5678"

    Print a.value(0)
    Print a.value("1")
    Print a.value(2)
    Print a.value("3")

    Sleep

```

This simulates an integer array that can be assigned strings, and even
 See KeyPgProperty for another example.

Restricting member access to certain parts of code.

Overview

Public members

Protected members

Private members

Constructors and destructors

Inherited members

Overview

All members of a **Type** or **Class** - including member data, procedures, constants, etc. - belong in one of three different classifications, each with its own rules dictating where in code they may be accessed, or referred to. These rules are called access rights. There are public, protected and private members, and they are declared in a **Type** or **Class** definition following a **Public**, **Protected** or **Private** label, respectively.

By default, that is, without an access classification label, members of **Type** are public, and members of a **Class** are private.

Public members

Public members can be referred to from anywhere; they are accessible from, for example, member procedures or module-level code or procedures.

Protected members

Protected members can only be accessed from member procedures of the **Type** or **Class** they are declared in, or member procedures of a derived **Type** or **Class**. They are not accessible to outside code.

Private members

Private members can only be accessed from member procedures of

the **Type** or **Class** they are declared in. They are not accessible to outside code or member procedures from a derived **Type** or **Class**.

Constructors and destructors

Constructors and destructors follow the same rules as any other member. When public, objects can be instantiated and destroyed from anywhere in code. When protected, objects can be instantiated and destroyed only from member procedures of their **Type** or **Class** or a derived **Type** or **Class**. Private constructors and destructors restrict object instantiation solely to member procedures of their **Type** or **Class**

Inherited members

...

Operator Overloading



Changing the way user defined types work with built-in operators.

Overview

Global Operators

Member Operators

Overview

Simply, operators are procedures, and their arguments are called *operands*. Operators that take one operand (**operator Not**) are called *unary operators*, operators that take two operands are called *binary operators* and operators taking three operands (**operator Mod**) are called *ternary operators*.

Most operators are not called like procedures. Instead, their operator symbol is placed between their operands. For unary operators, their sole operand is placed to the right of the operator symbol. For binary operators, their operands - referred to as the left and right-hand side operands - are placed to the left and right of the operator symbol. FreeBASIC has one ternary operator like a procedure, with its operands comma-separated surrounded by parentheses. The following code calls **operator Iif** to determine if a pointer is valid. If it is valid, it is called to dereference the pointer, and if not, **operator / (Divide)** is called to divide the pointer value by four.

```
Dim i As Integer = 420
Dim p As Integer Ptr = @i

Dim result As Integer = Iif( p, *p, CInt( 20 / 4 ) )
```

Notice the call to **operator Iif** is similar to a procedure call, while the call to **operator / (Divide)** is not. In the example, *p* is the operand to **operator Iif** and *20* and *4* are the left and right-hand side operands of **operator / (Divide)**.

All operators in FreeBASIC are predefined to take operands of standard types (**single**), but they may also be overloaded for user-defined types; that is, they can take operands that are objects as well. There are two types of operators: *global operators* and *member operators*.

Global Operators

Global operators are those that are declared in module-level scope (global scope). The following table lists the global operators and their meanings:

-	(Negate)
Not	(Bitwise Not)
->	(Pointer To Member Access)
*	(Multiply)
/	(Divide)
\	(Integer Divide)
&	(Concatenate)
Mod	(Modulus)
<<	(Shift Left)
>>	(Shift Right)
And	(Bitwise And)
Or	(Bitwise Or)
Xor	(Bitwise Xor)
Eqv	(Bitwise Eqv)
^	(Exponentiate)
=	(Equal)
<>	(Not Equal)
<	(Less Than)
<=	(Less Than Or Equal)
>	(Greater Than)
>=	(Greater Than Or Equal)

Declaring a custom global operator is similar to declaring a procedure with the `operator` keyword. The operator symbol is placed next to the procedure name, followed by parameters surrounded in parenthesis that will represent the operands. Operators can be overloaded by default, so the `Overload` keyword is used when declaring custom operators. At least one of the operator's parameters must be of a user-defined type (after all, operators with built-in type parameters are already defined).

The following example declares the global operators `-` (Negate) and `*` (Multiply) for a user-defined type.

```
Type Rational
    As Integer numerator, denominator
End Type

Operator - (ByRef rhs As Rational) As Rational
    Return Type(-rhs.numerator, rhs.denominator)
End Operator

Operator * (ByRef lhs As Rational, ByRef rhs As Rational) As Rational
    Return Type(lhs.numerator * rhs.numerator, _
                lhs.denominator * rhs.denominator)
End Operator

Dim As Rational r1 = (2, 3), r2 = (3, 4)
Dim As Rational r3 = -(r1 * r2)
Print r3.numerator & "/" & r3.denominator
```

Here the global operators are defined for type *Rational*, and are used *r3*. The output is *-6/12*.

Member Operators

Member operators are declared inside a **Type** or **Class** definition, like the cast and assignment operators **Let (Assign)**, **Cast (Cast)**, **+= (Add And Assign)**, ***= (Multiply And Assign)**, **/= (Divide And Assign)**, **\= (Integer Exponentiate And Assign)**, **&= (Concat And Assign)**, **Mod= (Modulus And Assign)**, **Shr= (Shift Right And Assign)**, **And= (Conjunction And Assign)**, **Xor= (Exclusive Disjunction And Assign)**, and **Eqv= (Equivalence And Assign)**.

When declaring member operators, the **Declare** and **Operator** keywords are used to declare the operator symbol and its parameter list. Like member procedures, member operators are declared outside the **Type** or **Class** definition, and the symbol name is prefixed with the name.

The following example overloads the member operators **Cast (Cast)** and ***=** for objects of a user-defined type.

```
Type Rational
  As Integer numerator, denominator

  Declare Operator Cast () As Double
  Declare Operator Cast () As String
  Declare Operator *= (ByRef rhs As Rational)
End Type

Operator Rational.cast () As Double
  Return numerator / denominator
End Operator

Operator Rational.cast () As String
  Return numerator & "/" & denominator
End Operator

Operator Rational.*= (ByRef rhs As Rational)
```

```
    numerator *= rhs.numerator
    denominator *= rhs.denominator
End Operator

Dim As Rational r1 = (2, 3), r2 = (3, 4)
r1 *= r2
Dim As Double d = r1
Print r1, d
```

Notice that the member operator **cast** (**Cast**) is declared twice, once for the conversion to **String**. This is the only operator (or procedure) that is declared multiple times when only the return type differs. The compiler decides which overload to use based on the type of the object being used. In the initialization of the **Double** *d*, *Rational.Cast as Double* is used. In the **Print** statement, *Rational.Cast as string* is used instead).

Types as Objects



An example of the overloadable operators and member procedures

Description

!!! WRITE ME !!!

```
' ' Sample Type showing available methods and opera
' ' Practically this is a pointless example, as the
' ' data member is an Integer. It serves only as a
' ' demonstration and guide.
' '
' ' There are many other combinations that can be
' ' used in pass parameters. For simplicity
' ' This example only uses byref and type T
' ' where ever possible.

' ' The type 'DataType' is included to show where
' ' any data type might be used
Type DataType As Integer

' ' The type 'UDT' is included to show where only
' ' a UDT data type can be used
Type UDT
    value As DataType
End Type

' ' Our main type
Type T
    value As DataType
    value_array( 0 ) As DataType

' ' let, cast, combined assignment operators,
' ' constructors, and the destructor, must be
' ' declared inside the type.
' '
' ' Parameters can be passed ByVal or Byref
```

```

'' in most (All? - verify this).
''
'' All procs can be overloaded with different
'' types as parameters. In many cases this is r
'' necessary as the TYPE can be coerced and
'' converted depending on the CAST methods
'' it exposes. The compiler will to its best
'' to evaluate statements and expressions if
'' there is enough information to complete
'' the operation.
''
'' For example,
'' Even though operator += may not be overloaded
'' but operator let and operator + are, the
'' compiler will convert the T += datatype
'' to T = T + datatype.

'' Nonstatic members must be declared inside the
'' type.
''
'' All Nonstatic members are implicitly
'' passed a hidden **this** parameter having
'' the same type as the TYPE in which they are
'' declared.
''
'' Nonstatic member overloaded operators do not
'' return a type. All operations are done on th
'' hidden this parameter.
''
'' Properties: Can be value properties or single
'' indexed value properties
'' GET/SET methods must be each delclared if used

'' Nonstatic Member Declarations:

'' Assignment

Declare Operator Let ( ByRef rhs As T )
Declare Operator Let ( ByRef rhs As DataType )

```

```

'' Cast can be overloaded to return multiple typ

Declare Operator Cast () As String
Declare Operator Cast () As DataType

'' Combined assignment

Declare Operator += ( ByRef rhs As T )
Declare Operator += ( ByRef rhs As DataType )

Declare Operator -= ( ByRef rhs As DataType )
Declare Operator *= ( ByRef rhs As DataType )
Declare Operator /= ( ByRef rhs As DataType )
Declare Operator \= ( ByRef rhs As DataType )
Declare Operator Mod= ( ByRef rhs As DataType )
Declare Operator Shl= ( ByRef rhs As DataType )
Declare Operator Shr= ( ByRef rhs As DataType )
Declare Operator And= ( ByRef rhs As DataType )
Declare Operator Or= ( ByRef rhs As DataType )
Declare Operator Xor= ( ByRef rhs As DataType )
Declare Operator Imp= ( ByRef rhs As DataType )
Declare Operator Eqv= ( ByRef rhs As DataType )
Declare Operator ^= ( ByRef rhs As DataType )

'' Address of

Declare Operator @ () As DataType Ptr

'' Constructors can be overloaded

Declare Constructor()
Declare Constructor( ByRef rhs As T )
Declare Constructor( ByRef rhs As DataType )

'' There can be only one destructor

Declare Destructor()

```

```

'' Nonstatic member functions and subs
'' overloaded procs must have different parameters

Declare Function f( ) As DataType
Declare Function f( ByRef arg1 As DataType ) As

Declare Sub s( )
Declare Sub s( ByRef arg1 As T )
Declare Sub s( ByRef arg1 As DataType )

'' Properties

Declare Property p ( ) As DataType
Declare Property p ( ByRef new_value As DataType

Declare Property pidx ( ByVal index As DataType
Declare Property pidx ( ByVal index As DataType,

End Type

'' These must be global procedures
'' Globals are not prefixed with the the TYPE name

'' At least one parameter must be of Type 'T'
'' For simplicity, type 'T' is always given first
'' in this example

Declare Operator - ( ByRef rhs As T ) As DataType
Declare Operator Not ( ByRef rhs As T ) As DataType

Declare Operator -> ( ByRef rhs As T ) As UDT
Declare Operator * ( ByRef rhs As T ) As DataType

Declare Operator + ( ByRef lhs As T, ByRef rhs As
Declare Operator - ( ByRef lhs As T, ByRef rhs As
Declare Operator * ( ByRef lhs As T, ByRef rhs As
Declare Operator / ( ByRef lhs As T, ByRef rhs As
Declare Operator \ ( ByRef lhs As T, ByRef rhs As
Declare Operator Mod ( ByRef lhs As T, ByRef rhs As A

```

```
Declare Operator Shl ( ByRef lhs As T, ByRef rhs As T )
Declare Operator Shr ( ByRef lhs As T, ByRef rhs As T )
Declare Operator And ( ByRef lhs As T, ByRef rhs As T )
Declare Operator Or ( ByRef lhs As T, ByRef rhs As T )
Declare Operator Xor ( ByRef lhs As T, ByRef rhs As T )
Declare Operator Imp ( ByRef lhs As T, ByRef rhs As T )
Declare Operator Eqv ( ByRef lhs As T, ByRef rhs As T )
Declare Operator ^ ( ByRef lhs As T, ByRef rhs As T )
Declare Operator = ( ByRef lhs As T, ByRef rhs As T )
Declare Operator <> ( ByRef lhs As T, ByRef rhs As T )
Declare Operator < ( ByRef lhs As T, ByRef rhs As T )
Declare Operator > ( ByRef lhs As T, ByRef rhs As T )
Declare Operator <= ( ByRef lhs As T, ByRef rhs As T )
Declare Operator >= ( ByRef lhs As T, ByRef rhs As T )
```

```
' ' Global procedures (subs and funcs) can also accept
' ' as a parameter or return it as a value, as could
' ' in previous versions of FreeBASIC.
' ' No example given. See function or sub in the manual.
```

```
' ' All TYPE members are defined outside the TYPE
```

```
' ' Nonstatic members must be prefixed with type name
' ' in this case 'T'
```

```
' ' Name resolution in a NAMESPACE is same as other
' ' subs/funcs. Use USING or prefix the namespace
```

```
Operator T.let ( ByRef rhs As T )
    value = rhs.value
End Operator
```

```
Operator T.let ( ByRef rhs As DataType )
    value = rhs
End Operator
```

```
Operator T.cast ( ) As String
    Return Str( value )
End Operator
```

```
Operator T.cast ( ) As DataType  
    Return value  
End Operator
```

```
Operator T.+= ( ByRef rhs As T )  
    value += rhs.value  
End Operator
```

```
Operator T.+= ( ByRef rhs As DataType )  
    value += rhs  
End Operator
```

```
Operator T.-= ( ByRef rhs As DataType )  
    value -= rhs  
End Operator
```

```
Operator T.*= ( ByRef rhs As DataType )  
    value *= rhs  
End Operator
```

```
Operator T./= ( ByRef rhs As DataType )  
    value /= rhs  
End Operator
```

```
Operator T.\= ( ByRef rhs As DataType )  
    value \= rhs  
End Operator
```

```
Operator T.mod= ( ByRef rhs As DataType )  
    value Mod= rhs  
End Operator
```

```
Operator T.shl= ( ByRef rhs As DataType )  
    value shl= rhs  
End Operator
```

```
Operator T.shr= ( ByRef rhs As DataType )  
    value shr= rhs
```

```

End Operator

Operator T.and= ( ByRef rhs As DataType )
    value And= rhs
End Operator

Operator T.or= ( ByRef rhs As DataType )
    value Or= rhs
End Operator

Operator T.xor= ( ByRef rhs As DataType )
    value Xor= rhs
End Operator

Operator T.imp= ( ByRef rhs As DataType )
    value Imp= rhs
End Operator

Operator T.eqv= ( ByRef rhs As DataType )
    value Eqv= rhs
End Operator

Operator T.^= ( ByRef rhs As DataType )
    value ^= rhs
End Operator

Operator T.@ ( ) As DataType Ptr
    Return( Cast( DataType Ptr, @This ))
End Operator

'' Constructors:

Constructor T()
    value = 0
End Constructor

Constructor T( ByRef rhs As T )
    value = rhs.value

```

```

End Constructor

Constructor T( ByRef rhs As DataType )
    value = rhs
End Constructor

'' There can be only one destructor

Destructor T()
    '' clean-up, none in this example
End Destructor

'' Globals must specify all arguments and return t

Operator - ( ByRef rhs As T ) As DataType
    Return (-rhs.value)
End Operator

Operator Not ( ByRef rhs As T ) As DataType
    Return (Not rhs.value)
End Operator

Operator -> ( ByRef rhs As T ) As UDT
    Return Type(4)
End Operator

Operator * ( ByRef rhs As T ) As DataType
    Return 5
End Operator

Operator + ( ByRef lhs As T, ByRef rhs As DataType
    Return (lhs.value + rhs)
End Operator

Operator - ( ByRef lhs As T, ByRef rhs As DataType
    Return (lhs.value - rhs)
End Operator

```

```
Operator * ( ByRef lhs As T, ByRef rhs As DataType  
    Return (lhs.value * rhs)  
End Operator
```

```
Operator / ( ByRef lhs As T, ByRef rhs As DataType  
    Return (lhs.value / rhs)  
End Operator
```

```
Operator \ ( ByRef lhs As T, ByRef rhs As DataType  
    Return (lhs.value \ rhs)  
End Operator
```

```
Operator Mod ( ByRef lhs As T, ByRef rhs As DataTy  
    Return (lhs.value Mod rhs)  
End Operator
```

```
Operator Shl ( ByRef lhs As T, ByRef rhs As DataTy  
    Return (lhs.value Shl rhs)  
End Operator
```

```
Operator Shr ( ByRef lhs As T, ByRef rhs As DataTy  
    Return (lhs.value Shr rhs)  
End Operator
```

```
Operator And ( ByRef lhs As T, ByRef rhs As DataTy  
    Return (lhs.value And rhs)  
End Operator
```

```
Operator Or ( ByRef lhs As T, ByRef rhs As DataTyp  
    Return (lhs.value Or rhs)  
End Operator
```

```
Operator Xor ( ByRef lhs As T, ByRef rhs As DataTy  
    Return (lhs.value Xor rhs)  
End Operator
```

```
Operator Imp ( ByRef lhs As T, ByRef rhs As DataTy  
    Return (lhs.value Imp rhs)
```

```
End Operator
```

```
Operator Eqv ( ByRef lhs As T, ByRef rhs As DataType  
    Return (lhs.value Eqv rhs)  
End Operator
```

```
Operator ^ ( ByRef lhs As T, ByRef rhs As DataType  
    Return (lhs.value ^ rhs)  
End Operator
```

```
Operator = ( ByRef lhs As T, ByRef rhs As DataType  
    Return (lhs.value = rhs)  
End Operator
```

```
Operator <> ( ByRef lhs As T, ByRef rhs As DataType  
    Return (lhs.value <> rhs)  
End Operator
```

```
Operator < ( ByRef lhs As T, ByRef rhs As DataType  
    Return (lhs.value < rhs)  
End Operator
```

```
Operator > ( ByRef lhs As T, ByRef rhs As DataType  
    Return (lhs.value > rhs)  
End Operator
```

```
Operator <= ( ByRef lhs As T, ByRef rhs As DataType  
    Return (lhs.value <= rhs)  
End Operator
```

```
Operator >= ( ByRef lhs As T, ByRef rhs As DataType  
    Return (lhs.value >= rhs)  
End Operator
```

```
'' Nonstatic member methods
```

```
Function T.f( ) As DataType  
    Dim x As DataType
```

```

Return x
End Function

Function T.f( ByRef arg1 As DataType ) As DataType
    arg1 = this.value
    Return value
End Function

Sub T.s( )
    '' refer to the type using

    '' with block
    With This
        .value = 1
    End With

    '' field access
    this.value = 2

    '' directly
    value = 3

End Sub

Sub T.s( ByRef arg1 As T )
    value = arg1.value
End Sub

Sub T.s( ByRef arg1 As DataType )
    value = arg1
End Sub

Property T.p ( ) As DataType
    '' GET property
    Return value
End Property

Property T.p ( ByRef new_value As DataType )
    '' SET property

```

```

    value = new_value
End Property

Property T.pidx ( ByVal index As DataType ) As Dat
    ' GET indexed property
    Return value_array( index )
End Property

Property T.pidx ( ByVal index As DataType, ByRef r
    ' SET indexed property
    value_array( index ) = new_value
End Property

' new, delete, delete[]

' Allocate object
Dim X As T Ptr = New T

' Deallocate object
Delete X

' Allocate object vector
Dim Xlist As T Ptr = New T[10]

' Deallocate object vector
Delete[] Xlist

```

See also

- [Type](#)

Control Flow Statements



Statements that direct the flow of execution.

Description

Control flow statements control program execution from one statement to the next; they determine what statements get executed and when, based on some kind of condition. The condition is always some expression that evaluates to true or false. Most control flow statements check for some kind of condition, and direct code flow accordingly, that is, they do or do not execute a block of code (except for the transferring control flow statements and **Do . . Loop**, which has an optional condition). Additionally, all control flow statements can be nested, that is, they can have other control flow statements within the statement block.

Control flow statements come in three flavors: transferring, branching and looping. Transferring control flow statements transfer execution to different parts of code. Branching control flow statements execute certain statements blocks based on a condition, while looping control flow statements execute code repeatedly while or until a condition is met.

Transferring Statements

These statements are used for either unconditional or conditional, temporary or permanent transfer of execution. The "ON" variants conditionally select a point of transfer from a list of text labels. Execution may be transferred between different scopes provided that the branching does not cross any local array, variable length string or object definition.

Goto

Unconditionally transfers execution to another point in code defined by a text label. Execution resumes with the first statement after the label.

GoSub

Unconditionally and temporarily transfers execution to another point in

code, defined by a text label. Execution resumes with the first statement after the label. Execution is then brought back to its original location with the **Return** keyword. Yes, **GoSub** statements can be nested, that is, multiple **GoSub** statements can be executed before the first corresponding **Return**, but there must always be a corresponding **Return** throughout the course of an application.

On Goto

Transfers execution to one of a number of points in code defined by text labels, based on the value of an expression.

On Gosub

Temporarily transfers execution to one of a number of points in code defined by text labels, based on the value of an expression.

Branching Statements

These statements are used for executing one of a number of statement blocks.

If..End If

Executes a block of statements if an expression evaluates to true (the condition). If and only if the expression evaluates to false, another statement block can be executed if yet another expression evaluates true using the **ElseIf** keyword. If and only if all of those expressions evaluate to false, a statement block can be executed using the **Else** keyword.

Select..End Select

Executes one of a number of statement blocks. This branching statement tries to meet a condition of an expression and one of a number of case expressions. The case expressions are checked in the order in which they are given, and the first case expression that is met has its associated statement block executed. Like **If..End If**, a default case can be defined when no other case expression meets the condition, and, as with the looping control flow statements, a case's statement block can be prematurely broken out of with the **Exit** keyword.

Looping Statements

These statements are used for executing a block of statements repeatedly. Within a statement block, the loop can be prematurely re-executed using the **continue** keyword, or broken out of using the **Exit** keyword. Whether the loop is terminated by the condition or with the **Exit** keyword, execution always begins at the first statement after the block.

While..Wend

Executes a block of statements while some expression evaluates to true (the condition). The expression is evaluated and checked before the block of statements is executed.

For..Next

Like **while..wend**, but more suited to loop a certain number of times. This loop initializes a so-called iterator with an initial value that is checked against a test expression. If the iterator compares less than or equal to the test expression (the condition), the block of statements is executed and the iterator gets incremented. The loop can also be setup so that the iterator gets decremented after every loop, in which case it is compared greater than or equal to the test expression. Iterators can be numeric data types like **Integer** or **Double**, or user-defined types. User-defined types must implement **Operator For**.

Do..Loop

The most versatile of the looping control flow statements, this loop can execute a block of statements while or until an expression evaluates to true (the condition). It can also delay the checking of the expression until after the block has executed the first time, useful when a block of statements needs to be executed *at least once*. Finally, this loop can have no condition at all, and merely loop indefinitely.

Procedures Overview



Overview of the different FB procedure types.

Procedures are blocks of code that can be executed, or called, from any number of times. The code that is executed is called the procedure body
procedures in FreeBASIC: procedures that don't return a value and proc

Subs

Subs are procedures that don't return values. They are declared using and defined using the `Sub` keyword. Declaring a procedure introduces its name and a procedure definition lists the statements of code that will be executed simply by using its name somewhere in the program.

```
' introduces the sub 'MyProcedure'  
Declare Sub MyProcedure  
  
' calls the procedure 'MyProcedure'  
MyProcedure  
  
' defines the procedure body for 'MyProcedure'  
Sub MyProcedure  
    Print "the body of MyProcedure"  
End Sub
```

will produce the output:

```
the body of MyProcedure
```

Notice that only the declaration is needed to call the procedure. The procedure is defined in code, or even in a different source file altogether.

Functions

Functions are procedures that return a value back to the point in code. You can think of a *function* call as evaluating to some expression, just like a variable declared using the **Declare** keyword, and defined using the **Function** keyword. The value that *functions* return is specified at the end of the declaration.

```
' introduces and defines a procedure that returns
Function MyProcedure As Integer
    Return 10
End Function

' calls the procedure, and stores its return value
Dim i As Integer = MyProcedure
Print i
```

will produce the output:

```
10
```

Since a definition is a declaration, a procedure can be called after it has been defined.

It is a common convention when calling a procedure to place parentheses around the name, to signify a procedure call. FreeBASIC does not require this, however.

See also

- **Passing Arguments to Procedures**
- **Returning a Value**
- **Declare**
- **Sub**
- **Function**

Passing Arguments to Procedures



Passing information to procedures.

Declaring parameters

Procedures can get passed information in the form of variables and objects. In the context of a procedure call, these variables and objects are called arguments. These arguments are then represented as so-called parameters inside the procedure. Parameters can be used just like any other variable or object.

To specify that a procedure should get passed arguments when called, you use a parameter list. A parameter list is a list of one or more names that a procedure will use when referring to the arguments that are passed to it, and it is surrounded by parenthesis.

```
Sub Procedure (s As String, n As Integer)
    Print "The parameters have the values: " & s & n
End Sub

Procedure "abc", 123
```

will produce the following output:

```
The parameters have the values: abc and 123
```

There are two ways to pass arguments to procedures: by value and by reference. By default, arguments are passed by value unless otherwise specified.

Passing arguments by value

Arguments that are passed by value are not actually passed to the procedure. Instead, a copy of the argument is made and passed instead. This allows the procedure to use the argument without changing the original variable or object remains unchanged.

When passing objects to procedures by value, the copy is made by calling the `Copy` method of the object.

constructor of the **Type** or **Class**.

To specify that an argument should be passed by value, precede the procedure declaration with the **ByVal** keyword:

```
Sub Procedure (ByVal param As Integer)
    param *= 2
    Print "The parameter 'param' = " & param
End Sub

Dim arg As Integer = 10
Print "The variable 'arg' before the call = " & arg
Procedure(arg)
Print "The variable 'arg' after the call = " & arg
```

will produce the following output:

```
The variable 'arg' before the call = 10
The parameter 'param' = 20
The variable 'arg' after the call = 10
```

Notice how parenthesis surround the arguments - in this case only on call. These parenthesis are optional, but are a common convention to

Passing arguments by reference

Unlike arguments that are passed by value, arguments that are passed by reference really do get passed; no copy is made. This allows the procedure to modify the original variable or object that was passed to it.

A reference is like an alias for a variable or object. Whenever you refer to a variable or object, you are referring to the actual variable or object that it aliases. In other words, a reference parameter of a procedure as the argument that is passed to the procedure and any changes to the reference parameter are actually changes to the argument it refers to.

To specify that an argument should be passed by reference, precede the procedure declaration with the **ByRef** keyword:

```

Sub Procedure (ByRef param As Integer)
    param *= 2
    Print "The parameter 'param' = " & param
End Sub

Dim arg As Integer = 10
Print "The variable 'arg' before the call = " & arg
Procedure(arg)
Print "The variable 'arg' after the call = " & arg

```

will produce the following output:

```

The variable 'arg' before the call = 10
The parameter 'param' = 20
The variable 'arg' after the call = 20

```

Manually passing pointers to by-reference parameters

By specifying the `ByVal` keyword in front of an argument to a `ByRef` parameter (usually stored in a pointer) can be passed directly as-is, forcing the parameter to reference the same memory location which the address pointed to.

```

Sub f( ByRef i As Integer )
    i = 456
End Sub

Dim i As Integer = 123
Dim pi As Integer Ptr = @i

Print i
f( ByVal pi )
Print i

```

See also

- **Procedures Overview**
- **Returning a Value**
- **Declare**
- **Sub**
- **Function**
- **ByVal**
- **ByRef**

Returning Values



Returning Values

... refers to the ability of a **Function** procedure to have a value when the expression is evaluated or assigned to a variable.

The value of a function can be returned in three ways:

```
' ' Using the name of the function to set the return value
Function myfunc1() As Integer
    myfunc1 = 1
End Function

' ' Using the keyword 'Function' to set the return value
Function myfunc2() As Integer
    Function = 2
End Function

' ' Using the keyword 'Return' to set the return value
Function myfunc3() As Integer
    Return 3
End Function
```

```
' ' This program demonstrates a function returning a value
Declare Function myFunction () As Integer

Dim a As Integer

'Here we take what myFunction returns and add 10.
a = myFunction() + 10

'knowing that myFunction returns 10, we get 10+10=20
```

```
Print a
```

```
Function myFunction ( ) As Integer  
    'Here we tell myFunction to return 10.  
    Function = 10  
End Function
```

Returning References

Function results can also be returned by reference, rather than by value.

When assigning a Byref function result through a Function = variable, the function returns the variable's value. Instead, it returns a reference to that variable. The reference returned from the function, without having to use pointers or arrays.

For more information, refer to: [Byref \(Function Results\)](#)

Manually returning pointers as-is from Byref functions

By specifying the ByVal keyword in front of the result variable in the Function (usually stored in a pointer) can be passed directly as-is, forcing the function to return the address pointed to. For example:

```
Dim Shared i As Integer = 123  
  
Function f( ) ByRef As Integer  
    Dim pi As Integer Ptr = @i  
  
    Function = ByVal pi  
  
    ' or, with RETURN it would look like this:  
    Return ByVal pi  
End Function  
  
Print i, f( )
```

See also

- **Function**
- **Byref (Function Results)**

Specifying how procedures are called.

Calling conventions determine how calling code interacts with procedure when called. They specify rules about how parameters are pushed onto the call stack, how values are returned and when the call stack is cleaned up. This information is useful when interacting with code written in other languages, particularly assembly language. In some cases, calling conventions also apply some kind of name decoration to procedure names.

FreeBASIC supports 3 calling conventions: **stdcall**, **cdecl** and **pascal**, specified with `stdcall`, `cdecl` and `pascal`, respectively. Calling convention can be specified in either a procedure declaration or definition immediately following the procedure name. The declaration of a procedure must have the same calling convention as the definition.

In all calling conventions, integral procedure return values are returned in the EAX(, EDX) register(s), and floating-point return values are stored in the ST(0) register (the top of the floating-point stack). User-defined type (UDT) values are returned in the EAX(, EDX) register(s) if eight (8) bytes or smaller, otherwise they are returned in memory by having their address pushed onto the call stack after any parameters.

stdcall

In the **stdcall** convention, procedure parameters are pushed onto the call stack prior to the procedure call (which will push the return address just above parameters) in the reverse order they are declared, that is, from right to left. The procedure is in charge of popping any parameters from the call stack (commonly by appending a constant to the RET instruction, signifying the number of bytes to release).

stdcall is the default calling convention on Windows, and for procedures within `Extern "Windows"` and `Extern "Windows-Ms"` blocks. It is also the default convention used in the Windows API.

Platform Differences

- In DOS and Windows platforms, the procedure name is decorated with an "@N" suffix, where *N* is the total size, in bytes, of any parameters passed.

cdecl

In the **cdecl** convention, procedure parameters are pushed onto the call stack prior to the procedure call, in the reverse order they are declared, that is, from right to left. The calling code is in charge of popping parameters from the call stack.

cdecl is the default calling convention on Linux, the *BSDs, and DOS and for procedures within **Extern "C"** and **Extern "C++"** blocks. It is also the default convention used by most C and C++ compilers.

pascal

In the **pascal** convention, procedure parameters are pushed onto the call stack, in the order they are declared, that is, from left to right. The procedure is in charge of popping any parameters from the call stack.

pascal is the default convention used by Pascal and the Microsoft QuickBASIC series of compilers.

The following table summarizes the differences between the various calling conventions:

Calling convention	Parameters are pushed onto the call stack from	Parameters are popped off the call stack by
stdcall	right to left	the procedure
cdecl	right to left	the calling code
pascal	left to right	the procedure

Platform Differences

- In DOS and Windows platforms, all calling conventions decorate procedure names with an underscore ("_") prefix.
- The default calling convention changes depending on the platform. For Windows it is **stdcall**; while on Linux, the *BSDs, and DOS, it is **cdecl**.

See also

- **Declare, Sub, Function**
- **stdcall, cdecl, pascal**
- **Extern..End Extern**

Pointers to Procedures



Pointers that point to procedures

Just as pointers can be made to point to an **Integer** or **Single** type, pointers can be made to point to a procedure.

Declaration

To declare a pointer to procedure, use the **Sub** or **Function** keywords, followed by the **As** keyword and the procedure name.

```
' declares a pointer to sub procedure that takes r
Dim pointerToProcedure As Sub
```

Procedure pointers store procedure addresses, which are retrieved using the **AddressOf** operator.

```
' pfunc.bi

Function Add (a As Integer, b As Integer) As Integer
    Return a + b
End Function

Dim pFunc As Function (As Integer, As Integer) As Integer
```

Calling a procedure pointer

The interesting thing about procedure pointers is that they can be called just like a procedure.

```
' .. Add and pFunc as before ..
#include once "pfunc.bi"

Print "3 + 4 = " & pFunc(3, 4)
```

For a calling example of subroutine pointer, see the **Operator @ (AddressOf)**.

Passing procedure pointers to procedures

Passing procedure pointers to other procedures is similar as well:

```
' ' .. Add and pFunc as before ..  
#include once "pfunc.bi"  
  
Function DoOperation (a As Integer, b As Integer,  
    Return operation(a, b)  
End Function  
  
Print "3 + 4 = " & DoOperation(3, 4, @Add)
```

Because procedure pointer declarations can be lengthy, it often helps

```
' ' .. Add and pFunc as before ..  
#include once "pfunc.bi"  
  
Type operation As Function (As Integer, As Integer  
  
Function DoOperation (a As Integer, b As Integer,  
    Return op(a, b)  
End Function  
  
Print "3 + 4 = " & DoOperation(3, 4, @Add)
```

Pointers to procedure pointers

Because the syntax of a procedure pointer does not allow declaration and not on procedure), a type alias is used. Notice how it is necessary This is because the function-call operator '()' has higher precedence t

```

Function Halve (ByVal i As Integer) As Integer
    Return i / 2
End Function

Function Triple (ByVal i As Integer) As Integer
    Return i * 3
End Function

Type operation As Function (ByVal As Integer) As Integer

' an array of procedure pointers, NULL indicates the
' end of the array
Dim operations(20) As operation = _
{ @Halve, @Triple, 0 }

Dim i As Integer = 280

' apply all of the operations to a variable by iterating
' with a pointer to procedure pointer
Dim op As operation Ptr = @operations(0)
While (*op <> 0)
    ' call the procedure that is pointed to, note
    i = (*op)(i)
    op += 1
Wend

Print "Value of 'i' after all operations performed"

```

Pointers to member procedures

Method pointers are not implemented yet, but it is possible to work-around

```

/''
' This example shows how you can simulate getting
' until support is properly implemented in the cc
'

```

```

' When this is supported, you will only need to r
' function presented here, to maintain compatibil
'/

Type T
    Declare Function test(ByVal number As Integer)
    Declare Static Function test(ByRef This As T,
    Dim As Integer i = 420
End Type

Function T.test(ByVal number As Integer) As Integer
    Return i + number
End Function

Function T.test(ByRef This As T, ByVal number As Integer) As Integer
    Return this.test(number)
End Function

Dim p As Function(ByRef As T, ByVal As Integer) As Integer
p = @T.test

Dim As T obj

Print p(obj, 69) '' prints 489

```

See also

- **Sub**
- **Function**
- **Pointer**
- **Operator @ (Address Of)**
- **Procptr Operator**

Variable Arguments



- ... (Ellipsis)
- va_arg
- va_first
- va_next

Static Libraries



A static library is compiled code that can be later used when building an

When the compiler makes an executable, the basic source files are first compiled into object files. These object files are then linked together to make an executable. When we compile a program, we have to make an executable. We could instead group all of the object files into a single file called a static library.

The library is referred to as static, because when the object files which it contains are linked into an executable, a copy of all the needed code in the library is added to the executable.

Once the library is made, we can then use the code that it contains just as if it were part of our program directly with our program.

Following is a simple example of creating a static library using these three files:

- `mylib.bas` - the source for the library
- `mylib.bi` - the header for the library
- `mytest.bas` - a test program

Our library will be a single module providing a single function:

```
' mylib.bas
' compile with: fbc -lib mylib.bas

' Add two numbers together and return the result
Public Function Add2( ByVal x As Integer, ByVal y As Integer )
    Return( x + y )
End Function
```

Compile the library with:

```
fbc -lib mylib.bas
```

The **-lib** option tells the compiler to take the source code, `mylib.bas`, and compile it into a library file, also called an archive, `libmy1.lib`. This library contains modules (source files) each with many functions, but for this simple example

To make use of the library in some other source code, we need some way to access what is in the library. A good way to do this is to put the declarations (also called symbols) in to a header file.

```
' ' mylib.bi
#include "mylib"
Declare Function Add2( ByVal x As Integer, ByVal y As Integer ) As Integer
```

There is no need to compile the header. We want this in its source form in our source files. The `#include` statement will tell the compiler the name of a header file when eventually making an executable.

With our library (.a file) and a header (.bi file) we can try them out in a test program.

```
' ' mytest.bas
' ' compile with: fbc mytest.bas
#include once "mylib.bi"
Print Add2(1,2)
```

The `#include` statement tells the compiler to include the source code from the original source. With the way we have written our include file, it tells the compiler to know about the library.

We compile this with:

```
fbc mytest.bas
```

Then when we run the `mytest` executable, we should get the result of:

3

More than one source module can be used when making a library. And you can make one library by including each needed header. Some libraries are so large that you can't include them all. On very large projects, making libraries out of some code modules that are used many times dramatically.

Libraries can optionally contain debugging information specified with the

Object files, and therefore libraries, are platform specific and in some cases the compiler and FreeBASIC runtime library.

See also

- **Shared Libraries**
- **`#inlib`**
- **`#include`**
- **Compiler Option: `-lib`**

A shared library is compiled code that can be loaded and used later when

When the compiler makes an executable, the basic source files are first
A shared library is much like a static library in that it contains object files
executable is running.

The library is referred to as shared, because the code in the library is loaded
though there might only be one copy of the shared library.

Once the library is made, we can then use the code that it contains just as

Shared Library Example
Using Shared Libraries on Windows
Using Shared Libraries on Linux
Executables that export symbols
Loading Shared Libraries Dynamically

Shared Library Example

Following is a simple example of creating a shared library using these

- `mylib.bas` - the source for the library
- `mylib.bi` - the header for the library
- `mytest.bas` - a test program

Our library will be a single module providing a single function:

```
' ' mylib.bas
' ' compile with: fbc -dll mylib.bas

' ' Add two numbers together and return the result
Public Function Add2( ByVal x As Integer, ByVal y
    Return( x + y )
End Function
```

Compile the library with:

```
fbc -dll mylib.bas
```

The `-dll` option tells the compiler to take the source code, `mylib.bas`, name of the shared library will have a `.so` extension or `.dll` extension many modules (source files) each with many functions, but for this sin

Making a shared library is almost identical to making a static library ex function visible to other executables loading the shared library.

To make use of the library in some other source code, we need some declarations (also called an interface, or API) for the library in to a he

```
' ' mylib.bi
#include "mylib"
Declare Function Add2( ByVal x As Integer, ByVal y
```

There is no need to compile the header. We want this in its source for compiler the name of a shared library that we need to link with at runti

With our library (`.dll` / `.so` file) and a header (`.bi` file) we can try them or

```
' ' mytest.bas
' ' compile with: fbc mytest.bas
#include once "mylib.bi"
Print Add2(1,2)
```

The `#include` statement tells the compiler to include the source code f written our include file, it tells the compiler everything it needs to know

We compile this with:

```
fbc mytest.bas
```

Then when we run the `mytest` executable, we should get the result of:
3

More than one source module can be used when making a library. An libraries are so large that they might use several headers. On very lar improve compile times and link times dramatically.

Shared libraries can optionally contain debugging information specifie

Object files, and therefore shared libraries, are platform specific and in library.

Using Shared Libraries on Windows

On Windows, the shared library must be stored in a location where it c

The operating system may search the following directories:

- The directory from which the executable was loaded.
- The current directory.
- The Windows and Windows system folder.
- Directories list in the `PATH` environment variable.

The order in which directories are searched may depend on the Wind

Using Shared Libraries on Linux

By default, Linux will not normally search the current directory or the c

- copy the `.so` file to a directory that has shared libraries (e.g. `/u`
- modify the environment variable `LD_LIBRARY_PATH` to search

To run the executable `./mytest/` and temporarily tell linux to search th

```
LD_LIBRARY_PATH=./:$LD_LIBRARY_PATH ./mytest
```

Executables that export symbols

If an executable has symbols that must be available to other shared libraries, use the `Export` specifier, and the `-export` command line option when making (linking)

The `-export` option has no extra effect when used with the `-dylib` or `-c`

Loading Shared Libraries Dynamically

Shared libraries can be loaded and used at run time by dynamically loading

- `DyLibLoad` can be used to load and obtain a handle to a shared library
- `DyLibSymbol` is used to obtain the address of a symbol in a loaded library
- `DyLibFree` is used to unload a shared library when it is no longer needed

Procedures in the shared library must use the `Export` specifier to ensure

```
' ' mydll.bas
' ' compile as: fbc -dll mydll.bas
' ' This will create mydll.dll (and libmydll.dll.a
' ' and libmydll.so on Linux.
' '
' ' Note: libmydll.dll.a is an import library, it's
' ' an executable that calls any of mydll's functions
' ' the DLL files with your apps, do not include them
' ' they are useless to end-users.
' '
' ' Simple exported function; the <alias "..."> disables
' ' all-upper-case name mangling, so the DLL will export
' ' ADDNUMBERS().
Function AddNumbers Alias "AddNumbers"( ByVal a As Integer, ByVal b As Integer) As Integer
    Function = a + b
End Function
```

```

'' load.bas: Loads mydll.dll (or libmydll.so) at runtime
'' functions and prints the result. mydll is not required
'' compile as: fbc test.bas
''
'' Note: The compiled mydll.dll (or libmydll.so) must be
'' to be available in the current directory.

'' Note we specify just "mydll" as library file name for
'' compatibility between Windows and Linux, where the latter
'' has different file name and extension.
Dim As Any Ptr library = DyLibLoad( "mydll" )
If( library = 0 ) Then
    Print "Failed to load the mydll dynamic library"
    End 1
End If

'' This function pointer will be used to call the function
'' the address has been found. Note: It must have the correct
'' convention and parameters.
Dim AddNumbers As Function( ByVal As Integer, ByVal As Integer )
AddNumbers = DyLibSymbol( library, "AddNumbers" )
If( AddNumbers = 0 ) Then
    Print "Could not retrieve the AddNumbers() function"
    End 1
End If

Randomize Timer

Dim As Integer x = Rnd * 10
Dim As Integer y = Rnd * 10

Print x; " +"; y; " ="; AddNumbers( x, y )

'' Done with the library; the OS will automatically free it
'' by a process when it terminates, but we can also free it
'' our program execution to save resources; this is optional

```

```
' ' Remember that once you unload a previously load  
' ' you got from it via dylibsymbol will become inv  
' ' will cause the application to crash.  
DyLibFree( library )
```

See also

- **Static Libraries**
- **#inlib**
- **#include**
- **Compiler Option: -dll**
- **Compiler Option: -export**
- **Compiler Option: -dylib**

Profiling can be used to analyze the performance of an application.

The performance of an application might be measured by how many times functions are called, how much time is spent executing those functions, and which functions are calling other functions. This can help to identify functions that might be taking too long to execute or executed too many times and that might be worth reviewing for optimization.

FreeBASIC uses GPROF for analyzing the execution of an application. The profiler information is collected while the program is running, and GPROF is used to report on the collected data afterward.

The three basic steps to profiling a program are:

- 1) Prepare the program for profiling by compiling source with the *profile* option.
- 2) Run the program to collection information (stored in `gmon.out`)
- 3) Analyze the information collected using GPROF.

Full documentation on GPROF is available here:

<http://www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html>

If the documentation has moved from that location, simply search the web for "GNU GPROF" and a relevant link should be returned.

FreeBASIC supports function profiling; not basic-block or line-by-line profiling.

Preparing a Program for Profiling

Only code that is compiled with the *-profile* command line option can be profiled. Pass the *-profile* option to the FreeBASIC compiler to prepare the program to be profiled. This will tell the compiler to insert special startup code at the beginning of the application as well as at the beginning of each function.

```
fbc program.bas -profile
```

Profiling the Program

The information needed to analyze execution of the program is gathered while the program is running. Run the program to begin collecting the function call information. This information is automatically stored in a file named `gmon.out` in the same directory as the program.

Analyzing the Program's Output

Use GPROF to analyze the output. The default report for GPROF includes descriptions on what each of the columns of values mean. If you are new to using GPROF, you may want to first run the default report and read through the descriptions. The output from GPROF can be saved to a file by redirection.

Save output from GPROF to `profile.txt`:

```
gprof program[.exe] > profile.txt
```

Show just the flat report with no descriptions:

```
gprof program[.exe] --brief --flat-profile > profile.txt
```

Combining the Results of More than One Session

GPROF also has a '--sum' option for conveniently combining results from multiple execution sessions. Here is an example of usable:

- Run your program once. This will create `gmon.out`.
- Use the command :

```
mv gmon.out gmon.sum
```

or

```
rename gmon.out gmon.sum.
```

- Run your program again. This will create new data in `gmon.out`.
- Merge the new data in `gmon.out` into `gmon.sum` using the command:

```
gprof --sum program[.exe] gmon.out gmon.sum
```

- Repeat the last two steps as often as needed.
- Analyze the summary data using the command:

```
gprof program[.exe] gmon.sum > profile.txt
```

FreeBASIC Profiling Internals

When the '-profile' option is enabled, one or more bits of code are added to the program.

- Call to "_monstartup()" at the beginning of the implicit main to initialize the profiling library.
- Call to "mcount()" at the beginning of each procedure. This is how the profiling library keeps track of what function is being and by which other function.
- Linking of additional program startup object code. (e.g. gcrt?.o)

The profiling library itself may be in a separate library or part of the C runtime library.

- mingw will require gcrt2.o and libgmon.a
- cygwin will require gcrt0.o and libgmon.a
- dos will require gcrt0.o (profiler code is in libc.a)
- linux will require gcrt1.o (profiler code is in libc.a)

The details may vary from one port of FreeBASIC to the next, but source code built for profiling with FreeBASIC should be compatible with other languages also supporting GPROF.

Table of ASCII Characters



FreeBASIC graphics programs support in all versions the same "ASCII extended" USA character set the old DOS (and QBasic) supported. It is also called CP437 or Code page 437. Each character is represented with one (1) byte of data. Here is a table. Each entry has decimal code, hex code, and printed representation.

```
000 00 |032 20 sp|064 40 @|096 60 `|128 80 Ç|160 A0 á|192 C0
L 224E0α 00101☉ 03321! 06541A 09761a 12981ü 161A1í 193C1↓
225E1ß 00202● 03422" 06642B 09862b 13082é 162A2ó 194C2↑ 226E2
00303♥ 03523# 06743C 09963c 13183â 163A3ú 195C3↓ 227E3π 00404
03624$ 06844D 10064d 13284ä 164A4ñ 196C4– 228E4Σ 00505♣
03725% 06945E 10165e 13385à 165A5Ñ 197C5† 229E5σ 00606♠
03826& 07046F 10266f 13486å 166A6ª 198C6‡ 230E6μ 00707• 03927'
07147G 10367g 13587ç 167A7º 199C7‖ 231E7τ 00808□ 04028( 07248l
10468h 13688ê 168A8¿ 200C8ℓ 232E8φ 00909○ 04129) 07349l 10569i
13789ë 169A9– 201C9⏚ 233E9θ 0100A■ 0422A* 0744AJ 1066Aj 1388A
170AA¬ 202CA± 234EAΩ 0110B♁ 0432B+ 0754BK 1076Bk 1398Bï
171AB½ 203CB⏚ 235EBδ 0120C♀ 0442C, 0764CL 1086Cl 1408Cî
172AC¼ 204CC‖ 236EC∞ 0130D, 0452D- 0774DM 1096Dm 1418Dì
173ADì 205CD= 237EDφ 0140E, 0462E. 0784EN 1106En 1428EÄ
174AE« 206CE‡ 238EEε 0150F✳ 0472F/ 0794FO 1116Fo 1438FÅ
175AF» 207CF± 239EF∩ 01610▶ 048300 08050P 11270p 14490É
176B0☼ 208D0ℓ 240F0≡ 01711◀ 049311 08151Q 11371q 14591æ
177B1☼ 209D1‡ 241F1± 01812↑ 050322 08252R 11472r 14692Æ
178B2☼ 210D2‡ 242F2≥ 01913!! 051333 08353S 11573s 14793ô 179B3
211D3ℓ 243F3≤ 02014¶ 052344 08454T 11674t 14894ö 180B4↓ 212D4
244F4† 02115§ 053355 08555U 11775u 14995ð 181B5↓ 213D5‡ 245F5
02216– 054366 08656V 11876v 15096û 182B6‖ 214D6‡ 246F6÷ 02317
055377 08757W 11977w 15197ù 183B7‡ 215D7‖ 247F7≈ 02418†
056388 08858X 12078x 15298ÿ 184B8‡ 216D8‡ 248F8° 02519↓ 05739!
08959Y 12179y 15399Ö 185B9‡ 217D9‡ 249F9· 0261A→ 0583A:
0905AZ 1227Az 1549AÜ 186BA‖ 218DA‡ 250FA• 0271B– 0593B;
0915B[ 1237B{ 1559Bç 187BB‡ 219DB■ 251FB√ 0281C_ 0603C<
0925C\ 1247C| 1569C£ 188BC‡ 220DC■ 252FCⁿ 0291D↔ 0613D=
0935D] 1257D} 1579D¥ 189BD‡ 221DD■ 253FD² 0301E▲ 0623E>
```

0945E^ 1267E~ 1589E_{rs} 190BE_l 222DE | 254FE■ 0311F▼ 0633F?
0955F_ 1277F△ 1599Ff 191BF_γ 223DF■ 255FF

Many of the standard ASCII characters cannot be **Printed** in FreeBASIC because the console interprets some characters as controls: 7 is bell, 8 is backspace, 9 is tab, 10 is line feed, 13 is carriage return, and others. There are symbols associated with these characters also, but there is no way in FreeBASIC to output them to the screen.

The acronym ASCII stands for American Standard Code for Information Interchange. For more information, see <http://en.wikipedia.org/wiki/Ascii>. The symbols for codes 32 through 127 are the same as the standard **Latin ISO-8859-1** char set most Windows fonts use. Others are often very different.

In console mode (i.e. **Screen 0** / non-graphics mode) the characters less than 32 or greater than 127 may display using different characters, depending on the operating system and code page of the screen / console in use.

UNICODE is a newer standard of character sets involving two or more bytes per character, and may be used to print other characters to a Unicode-enabled console.

In graphics modes, **Draw String** does not give special meaning to control characters allowing an alternative to display all characters in the set.

Description

A date serial is a number that holds a date and time value in the same used by PDS or VBDOS. The value is a count of the days from 0:00 A December 30,1899; it's mainly used for easy counting of the time betw dates.

The date serial unit is one day and the fractional part represents the ti day. If a date serial is written into an integer, the time is lost. FreeBAS serials are not limited to dates between 1753 and 2078 as in VBDOS. FreeBASIC date serial handling functions use **Double** arguments.

FreeBASIC date serial handling functions require the inclusion of `vbcc datetime.bi` in the source.

A date serial can be created by the functions **Now**, **TimeSerial+DateSer** **DateValue+TimeValue**.

The functions **Year**, **Month**, **Weekday**, **Day**, **Hour**, **Minute**, **Second** allow to r different components of a date serial.

The **Format** function has formatting expressions to print date serials in readable way.

Example

```
#include "vbcompat.bi"
Dim a As Double, b As Double

a = 0
Print "The origin of the date serials is:"
Print Format(a, "yyyy/mm/dd hh:mm:ss")
Print

a = Now
Print "The time now is: "
```

```
Print Format(a, "yyyy/mm/dd hh:mm:ss")  
Print
```

```
b = DateSerial(2000,1,1)
```

```
Print Int(a-b) & " days have passed since 2000/01/
```

Radian system of measuring angles



All of the built-in trigonometric functions in FreeBASIC express angles in radians.

A full circle is divided into $2 * \pi$ radians or 360 degrees, which leads to the following conversions:

```
radians = degrees * pi / 180  
degrees = radians * 180 / pi
```

Pi is a constant equal to the ratio of the circumference of a circle to its diameter. It can be calculated programmatically by multiplying the arctangent of 1 by 4:

```
pi = atn(1) * 4
```

Pixel formats

When a graphics mode is set via the `Screen` or `ScreenRes` functions, GfxLib creates also a framebuffer in standard system memory and sets an appropriate internal pixel format for the mode. There are basically three internal pixel formats, selected depending on the screen depth, as described in the following table:

Screen depth	Internal bytes per pixel	Range bitmask	Pixel format
1bpp	1	&h1	palette color index
2bpp	1	&h3	palette color index
4bpp	1	&hF	palette color index
8bpp	1	&hFF	palette color index
15/16bpp	2	&hFFFF	RRRRRGGGGGGBBBBB
24/32bpp	4	&hFFFFFFFF	AAAAAAAAARRRRRRRRGGGGGGGGBBBBBBBB

All drawing operations work on this RAM framebuffer; when the actual display needs to be updated, GfxLib copies the contents of the framebuffer to the real display memory, automatically converting in the process from the current internal pixel format to whatever pixel format the real display uses. By limiting the internal pixel formats to 3, the library prevents you having to deal with the plethora of real display formats.

Color values

When calling a graphics primitive that accepts a color, this can be specified in two ways. In 8bpp or less modes, the color value must be a direct 8 bits color index in the current palette, and this matches the internal pixel format for those modes. In higher color depths, the color value should always have the form `&hAARRGGBB;`; this is what the `RGB` and `RGBA` macros return, and is equivalent to the 24/32bpp internal pixel format representation. If the current color depth is 24 or 32bpp, this means the color value passes in unaltered. If a 15/16bpp mode is in use

internally each primitive automatically converts the color from the &hAARRGGBB; form into the RRRRRGGGGGGBBBBB internal pixel format (note that in this process the alpha channel is lost, as 15/16bpp modes do not support it). Once the color value is in one of the three pixel formats, the primitive limits its range to the range supported by the current color depth, by using a bitwise **And** operation with a range bitmask. So if in 8bpp, the color value passed is **Anded** by &hFF; for example.

Notes on transparency

For 8bpp or less modes, color index 0 is always treated as the transparent color for the **Put** modes that support transparency. For higher depths, **RGB**(255, 0, 255) always represents the transparent color. In 15/16bpp modes, this translates to the internal value &hF81F;, whereas in 24/32bpp modes it becomes &hFFFF00FF;. Note that in 24/32bpp modes, **Put** identifies the transparent color by looking just at the red, green and blue components of the color value, while the alpha value can assume any value. This means that in 24/32bpp modes, &h00FF00FF;, &h10FF00FF;, &hABFF00FF; and &hFFFF00FF; for example all represent the transparent color, since the lower 24 bits are always &hFF00FF;.

Buffer formats

In FreeBASIC, images can be used as arrays (as in QB) or as pointers. Either way, the image data is contained in one continuous chunk. The chunk consists of an header followed by the image data. The header can be of two types (old-style and new-style) and determines the format of the following image data.

Old-style chunk header consists of 4 bytes (32 bits, or 4 bytes). The first 3 bits contain the image color depth in bytes per pixel (8-bit color depth -> 1; 16-bit color depth -> 2; 32-bit color depth -> 4). The next 13 bits contain the image width. The last 16 bits contain the image's height. Please note the intrinsic nature of the header allows only for sizes up to 8191 * 65535 pixels. The actual pixel data follows the header, and is compacted one row of pixels after another; no data alignment is assumed. The final size of the chunk can then be computed using the formula:

```
size = 4 + ( width * height * bytes_per_pixel )
```

New-style chunk header consists of 32 bytes. The first dword (32 bits) must be equal to the value 7, allowing GfxLib to identify the new type of chunk. The second dword contains the image color depth in bytes per pixel. The third and fourth dwords contain the image width and height respectively, effectively removing the image size limit enforced by the old style image chunks. The fifth dword contains the pixel row pitch in bytes this tells how many bytes a row of pixels in the image takes up. The pitch in new-style chunks is always padded to a multiple of 16, to allow pixels' row data to be aligned on the paragraph boundary. The remaining 3 dwords (total 12 bytes) of the header are currently unused and reserved for future use. The final size of the image is:

```
size = 32 + ( ( ( ( width * bytes_per_pixel ) + &hF; ) and not &hF ) * height )
```

The format of images created by `ImageCreate` and `Get` depend on the dialect used. In the *-lang fb* dialect, images will be created with the new style header. In the *-lang fblite* and *-lang qb* dialects, the old-style image header is created.

All graphics primitives can work with both old-style and new-style image chunks. For easy access to image information, `ImageInfo` can be used to obtain useful properties of an image buffer - such as its dimensions, color depth, pitch, and a pointer to the pixel data - whichever format is used. It is also possible to access the image header directly to access this information. For more information on accessing the header structure, please refer to [this example](#).

See also

- [Screen \(Graphics\)](#)
- [ScreenRes](#)
- [Get \(Graphics\)](#)
- [Put \(Graphics\)](#)
- [ImageCreate](#)
- [ImageInfo](#)

- Trans
- Alpha

C Standard Library Functions



This is a list of function prototypes in the standard C library in alphabetical order and a list of prototypes grouped by functionality.

Alphabetical List
Buffer Manipulation
Character Classification and Conversion
Data Conversion
Directory Manipulation
File Manipulation
Stream I/O
Low level I/O
Mathematics
Memory Allocation
Process Control
Searching and Sorting
String Manipulation
Time

Description

The Comments column contains a very brief description of the use of the function. The list is not complete, however it provides information on the major functions in the C Runtime Library. It should, at the very least, indicate what functions are available in the standard C library allow you to do more investigation on your own. Some of the C library functions documented elsewhere may not be available in FreeBASIC. Check the appropriate include file for more information.

Note: The following prototypes are not the official FreeBASIC prototypes (see the include files), however, they will give you enough information to properly use the functions.

The Include File column contains the name of the file which you must include, using the `#include` directive at the beginning of your program. If you don't include the appropriate include file, the program either will not compile, or it will compile apparently correctly but give incorrect

results when run. All of the C Runtime headers are located in the `crt` directory; for example, if the specified header is `math.bi`, use `#include "crt/math.bi"` or `#include "crt\math.bi"`, or just `#include "crt.bi"` including all the others.

The Prototype column contains the following information:

- The name of the function;
- The parameters required for the function in parenthesis, together with the data-type of the parameters;
- The data-type of the value returned by the function.

For example `atoi(a as zstring ptr) as integer` means that the function `atoi` returns a value of type **integer** and requires a character **zstring ptr** as its argument.

Note: In order to make calling the C runtime functions very easy, any string type argument may be directly passed to a procedure referring a parameter declared as 'zstring ptr'. The compiler performs itself an automatic conversion (without warning message) between string and 'zstring ptr'.

Alphabetical List

Name	Prototype (with parameters)	Include File	Comments
<code>abs_</code>	<code>abs_(n as integer) as integer</code>	<code>stdlib.bi</code>	Returns the absolute value (i.e. positive value)
<code>acos_</code>	<code>acos_(a as double) as double</code>	<code>math.bi</code>	Returns the inverse cosine (angle in radians)
<code>asin_</code>	<code>asin_(a as double) as double</code>	<code>math.bi</code>	Returns the inverse sine (angle in radians)
<code>atan_</code>	<code>atan_(a as double) as double</code>	<code>math.bi</code>	Returns the inverse tan (angle in radians)
<code>atan2_</code>	<code>atan2_(y as double, x as double) as double</code>	<code>math.bi</code>	Returns the inverse tan (pass the opposite as y and the adjacent as x)
<code>atoi</code>	<code>atoi(s as zstring ptr) as integer</code>	<code>stdlib.bi</code>	Converts a character zstring of digits to a number of type integer.

atof	atof(s as zstring ptr) as double	stdlib.bi	Converts a character zstring of digits to a number of type double.
calloc	calloc(NumElts as integer, EltSiz as integer) as any ptr	stdlib.bi	Allocates memory. Returns a pointer to a buffer for an array having NumElts elements, each of size EltSiz bytes.
ceil	ceil(d as double) as double	math.bi	Returns the nearest whole number above the value passed.
clearerr	clearerr(s as FILE ptr)	stdio.bi	Clears the error indicators on a file stream (read or write).
cos_	cos_(ar as double) as double	math.bi	Returns the cosine of an angle measured in radians.
cosh	cosh(x as double) as double	math.bi	Returns the hyperbolic cosine of an angle measured in radians.
div	div(num as integer, denom as integer) as div_t	stdlib.bi	Returns the quotient and remainder of a division as a structure of type div_t.
ecvt	ecvt(x as double) as zstring ptr	math.bi	Converts a number to a zstring.
exit_	exit_(status as integer)	stdlib.bi	Exits a program. It will flush file buffers and closes all opened files, and run any functions called by atexit().
exp_	exp_(a as double) as double	math.bi	Returns the value of e raised to the power of the argument (Inverse to natural logarithm).
fabs	fabs(d as double) as double	math.bi	Returns the absolute value (i.e. positive value) of type double.
fclose	fclose(s as FILE ptr) as FILE ptr	stdio.bi	Closes a file. Returns 0 if successful otherwise EOF.
feof	feof(s as FILE ptr) as integer	stdio.bi	Returns value of end-of-file indicator . (0 if not eof). Indicator will clear itself but can be reset by clearerr().
ferror	ferror(s as FILE ptr) as integer	stdio.bi	Returns error indicator for a stream (0 if no error). Error indicator is reset by clearerr() or rewind().
fflush	fflush(s as FILE ptr) as integer	stdio.bi	Flushes (i.e. deletes) a stream (use stdin to flush the stream from the keyboard). Returns 0 if successful.
fgetc	fgetc(s as FILE ptr) as integer	stdio.bi	Single character input (in ASCII) from passed stream (stdin for keyboard).
fgetpos	fgetpos(s as FILE ptr, c as fpos_t ptr) as integer	stdio.bi	Saves the position of the file pointer on stream s at the location pointed to by c.
fgets	fgets(b as zstring ptr, n as integer, s as FILE ptr) as zstring ptr	stdio.bi	From the stream s reads up to n-1 characters to buffer b.

floor	floor(d as double) as double	math.bi	Returns the nearest whole number below the value passed.
fmod	fmod(x as double, y as double) as double	math.bi	Calculates the remainder of x divided by y.
fopen	fopen(file as zstring ptr, mode as zstring ptr) as FILE ptr	stdio.bi	Opens a file. Pass the DOS name of the file and a code to indicate whether for reading, writing, or appending. Codes are r for read, w for write, + for read and write, a for append and b to indicate binary.
fprintf	fprintf(s as FILE ptr, fmt as zstring ptr, ...) as integer	stdio.bi	Prints on stream s as many items as there are single % signs in fmt that have matching arguments in the list.
fputc	fputc(c as integer, s as FILE ptr) as integer	stdio.bi	Outputs the single character c to the stream s.
fputs	fputs(b as zstring ptr, s as FILE ptr) as integer	stdio.bi	Sends the character stream in b to stream s, returns 0 if the operation fails.
fread	fread(buf as any ptr, b as size_t, c as size_t, s as FILE ptr) as integer	stdio.bi	Reads the number c items of data of size b bytes from file s to the buffer buf. Returns the number of data items actually read.
free	free(p as any ptr)	stdlib.bi	Frees the memory allocation for a pointer p to enable this memory to be used.
freopen	freopen(file as zstring ptr, mode as zstring ptr, s as FILE ptr) as FILE ptr	stdio.bi	Opens a file for redirecting a stream. e.g. freopen("myfile", "w", stdout) will redirect the standard output to the opened "myfile".
frexp	frexp(x as double, p as integer ptr) as double	math.bi	Calculates a value m so that x equals m times 2 to some power. p is a pointer to m.
fscanf	fscanf(s as FILE ptr, fmt as zstring ptr, ...) as integer	stdio.bi	Reads from stream s as many items as there are % signs in fmt with corresponding listed pointers.
fseek	fseek(s as FILE ptr, offset as integer, origin as integer) as integer	stdio.bi	Locates a file pointer. With origin 0, 1 or 2 for the beginning, offset bytes into and at the end of the stream.
fsetpos	fsetpos(s as FILE ptr, p as fpos_t ptr) as integer	stdio.bi	Sets the file pointer for the stream s to the value pointed to by p.
ftell	ftell(s as FILE ptr) as long	stdio.bi	Locates the position of the file pointer for the stream s.
	fwrite(buf as any		

fwrite	ptr, b as integer, c as integer, s as FILE ptr) as integer	stdio.bi	Writes the number c items of data of size b bytes from the buffer buf to the file s. Returns the number of data items actually written.
getc	getc(s as FILE ptr) as integer	stdio.bi	Macro for single character input (in ASCII) from passed stream. (stdin for keyboard)
getchar	getchar() as integer	stdio.bi	Single character input from the standard input
gets	gets(b as zstring ptr) as zstring ptr	stdio.bi	Reads a stream of characters from the standard input until it meets \n or EOF.
hypot	hypot(x as double, y as double) as double	math.bi	Calculates the hypotenuse from the sides x and y.
isalnum	isalnum(c as integer) as integer	ctype.bi	Returns a non zero value if character c is alphabetic or a digit.
isalpha	isalpha(c as integer) as integer	ctype.bi	Returns a non zero value if character c is alphabetic.
iscntrl	iscntrl(c as integer) as integer	ctype.bi	Returns a non zero value if character c is a control character.
isdigit	isdigit(c as integer) as integer	ctype.bi	Returns a non zero value if character c is a digit.
isgraph	isgraph(c as integer) as integer	ctype.bi	Returns a non zero value if character c is alphabetic.
islower	islower(c as integer) as integer	ctype.bi	Returns a non-zero value if character c is a lower case character.
isprint	isprint(c as integer) as integer	ctype.bi	Returns a non zero value if character c is printable.
ispunct	ispunct(c as integer) as integer	ctype.bi	Returns a non zero value if character c is a punctuation character.
isspace	isspace(c as integer) as integer	ctype.bi	Returns a non zero value if character c denotes a space.
isupper	isupper(c as integer) as integer	ctype.bi	Returns a non-zero value if character c is an upper case character.
isxdigit	isxdigit(c as integer) as integer	ctype.bi	Returns a non-zero value if character c is a hex digit (0 to F or f).
ldexp	ldexp(x as double, n as integer) as double	math.bi	Returns the product of x and 2 to the power n.
ldiv	ldiv(num as long, denom as long) as ldiv_t	stdlib.bi	Returns the quotient and remainder of a division as a structure of type ldiv_t.
log_	log_(a as double) as double	math.bi	Returns the natural logarithm of the argument.
log10	log10(a as double) as double	math.bi	Returns the logarithm to the base 10 of the argument.

malloc	malloc(bytes as integer) as any ptr	stdlib.bi	Allocates memory. Returns a pointer to a buffer comprising storage for the specified size.
modf	modf(d as double, p as double ptr) as double	math.bi	Returns the fractional part of a floating point number d. p points to the integral part expressed as a float.
perror	perror(mess as zstring ptr)	stdio.bi	Prints on the stream stderr a message passed as the argument.
pow	pow(x as double, y as double) as double	math.bi	Returns x to the power y.
pow10	pow10(x as double) as double	math.bi	Returns 10 to the power x (inverse function to log10()).
printf	printf(fmt as zstring ptr, ...) as integer	stdio.bi	Prints on standard output as many items as there are single % signs in fmt with matching arguments in the list.
putc	putc(c as integer, s as FILE ptr) as integer	stdio.bi	Macro to output the single character c to the stream s.
putchar	putchar(c as integer) as integer	stdio.bi	Macro to output the single character c to the standard output.
puts	puts(b as zstring ptr) as integer	stdio.bi	Sends the character stream in b to the standard output, returns 0 if operation fails.
rand	rand() as integer	stdlib.bi	Returns a pseudo random number. A seed is required. The seed is set with srand.
realloc	realloc(p as any ptr, newsize as size_t) as any ptr	stdlib.bi	Allocates memory. Returns a pointer to a buffer for a change in size of object pointed to by p.
rewind	rewind(s as FILE ptr)	stdio.bi	Clears the error indicators on a file stream (read or write). Necessary before reading an amended file.
scanf	scanf(fmt as zstring ptr, ...) as integer	stdio.bi	Reads from standard input as many items as there are % signs in fmt with corresponding listed pointers.
sin_	sin_(ar as double) as double	math.bi	Returns the sine of an angle measured in radians.
sinh	sinh(x as double) as double	math.bi	Returns the hyperbolic sine of an angle measured in radians.
sprintf	sprintf(p as zstring ptr, fmt as zstring ptr, ...) as integer	stdio.bi	Prints on zstring p as many items as there are single % signs in fmt that have matching arguments in the list.
sqrt	sqrt(a as double) as double	math.bi	Returns the square root of the value passed. Domain error if value is negative.
srand	srand(seed as uinteger)	stdlib.bi	Sets the seed for a random number. A possible seed is the current time.

sscanf	sscanf(b as zstring ptr, fmt as zstring ptr, ...) as integer	stdio.bi	Reads from buffer b as many items as there are % signs in fmt with corresponding listed pointers.
strcat	strcat(s1 as zstring ptr, s2 as zstring ptr) as zstring ptr	string.bi	Concatenates (appends) zstring s2 to s1.
strchr	strchr(s as zstring ptr, c as integer) as zstring ptr	string.bi	Returns a pointer to the first occurrence of c in s or NULL if it fails to find one.
strcmp	strcmp(s1 as zstring ptr, s2 as zstring ptr) as integer	string.bi	Compares zstring s2 to s1. Returns 0 or signed difference in ASCII values of first non matching character.
strcpy	strcpy(s1 as zstring ptr, s2 as zstring ptr) as zstring ptr	string.bi	Copies s2 into s1.
strcspn	strcspn(s1 as zstring ptr, s2 as zstring ptr) as integer	string.bi	Returns the number of characters in s1 encountered before meeting any of the characters in s2.
strerror	strerror(n as integer) as zstring ptr	string.bi	Returns a pointer to a system error message corresponding to the passed error number.
strlen	strlen(s as zstring ptr) as integer	string.bi	Returns the number of bytes in the null terminated zstring pointed to by s (does not count null).
strncat	strncat(s1 as zstring ptr, s2 as zstring ptr, n as integer) as zstring ptr	string.bi	Concatenates (appends) n bytes from zstring s2 to s1.
strncmp	strncmp(s1 as zstring ptr, s2 as any ptr, n as integer) as integer	string.bi	Compares n bytes of zstring s2 to the same of s1. Returns 0 or signed difference in ASCII values of first non matching character.
strncpy	strncpy(s1 as zstring ptr, s2 as zstring ptr, n as integer) as zstring ptr	string.bi	Copies n bytes from s2 into s1.
strpbrk	strpbrk(s1 as zstring ptr, s2 as zstring ptr) as zstring ptr	string.bi	Returns a pointer to the first character encountered in s1 that is also in s2.
strrchr	strrchr(s as zstring ptr, c as integer) as zstring ptr	string.bi	Returns a pointer to the last occurrence of c in s or NULL if it fails to find one.

strspn	strspn(s1 as zstring ptr, s2 as zstring ptr) as integer	string.bi	Returns the number of characters in s1 encountered before meeting a character which is not in s2.
strstr	strstr(s1 as zstring ptr, s2 as zstring ptr) as zstring ptr	string.bi	Finds the location of the zstring s2 in s1 and returns a pointer to its leading character.
strtod	strtod(s as zstring ptr, p as zstring ptr) as double	stdlib.bi	Converts a zstring to double, provided the zstring is written in the form of a number.
strtok	strtok(s1 as zstring ptr, s2 as zstring ptr) as zstring ptr	string.bi	Returns pointers to successive tokens utilizing the zstring s1. Tokens regarded as separators are listed in s2.
system	system(command as zstring ptr) as integer	stdlib.bi	Executes, from within a program, a command addressed to the operating system written as a zstring (e.g. DIR on Windows and DOS and LS on Linux).
tan_	tan_(ar as double) as double	math.bi	Returns the tangent of an angle measured in radians.
tanh	tanh(x as double) as double	math.bi	Returns the hyperbolic tangent of an angle measured in radians.
tolower	tolower(c as integer) as integer	ctype.bi	Converts a character from upper case to lower case (uses ASCII code).
toupper	toupper(c as integer) as integer	ctype.bi	Converts a character from lower case to upper case (uses ASCII code).
ungetc	ungetc(c as integer, s as FILE ptr) as integer	stdio.bi	Pushes a character c back into the stream s, returns EOF if unsuccessful. Do not push more than one character.

Buffer Manipulation

#include "crt/string.bi"

Prototype (with parameters)	Comments
memchr(s as any ptr, c as integer, n as size_t) as any ptr	Search for a character in a buffer.
memcmp(s1 as any ptr, s2 as any ptr, n as size_t) as integer	Compare two buffers.
memcpy(dest as any ptr, src as any ptr, n as size_t) as any ptr	Copy one buffer into another .
memmove(dest as any ptr, src as any ptr, n as	Move a number of bytes from one

size_t) as any ptr	buffer lo another.
memset(s as any ptr, c as integer, n as size_t) as any ptr	Set all bytes of a buffer to a given character.

Character Classification and Conversion

#include "crt/ctype.bi"

Prototype (with parameters)	Comments
isalnum(c as integer) as integer	True if c is alphanumeric.
isalpha(c as integer) as integer	True if c is a letter.
isascii(c as integer) as integer	True if c is ASCII .
isctrl(c as integer) as integer	True if c is a control character.
isdigit(c as integer) as integer	True if c is a decimal digit.
isgraph(c as integer) as integer	True if c is a graphical character.
islower(c as integer) as integer	True if c is a lowercase letter.
isprint(c as integer) as integer	True if c is a printable character.
ispunct(c as integer) as integer	True if c is a punctuation character.
isspace(c as integer) as integer	True if c is a space character.
isupper(c as integer) as integer	True if c is an uppercase letter.
isxdigit(c as integer) as integer	True if c is a hexadecimal digit.
toascii(c as integer) as integer	Convert c to ASCII .
tolower(c as integer) as integer	Convert c to lowercase.
toupper(c as integer) as integer	Convert c to uppercase.

Data Conversion

#include "crt/stdlib.bi"

Prototype (with parameters)	Comments
atof(string1 as zstring ptr) as double	Convert zstring to floating point value.

atoi(string1 as zstring ptr) as integer	Convert zstring to an integer value.
atol(string1 as zstring ptr) as integer	Convert zstring to a long integer value.
itoa(value as integer, zstring as zstring ptr, radix as integer) as zstring ptr	Convert an integer value to a zstring using given radix.
ltoa(value as long, zstring as zstring ptr, radix as integer) as zstring ptr	Convert long integer to zstring in a given radix.
strtod(string1 as zstring ptr, endptr as zstring ptr) as double	Convert zstring to a floating point value.
strtol(string1 as zstring ptr, endptr as zstring ptr, radix as integer) as long	Convert zstring to a long integer using a given radix.
strtoul(string1 as zstring ptr, endptr as zstring ptr, radix as integer) as ulong	Convert zstring to unsigned long.

Directory Manipulation

```
#include "crt/io.bi"
```

Prototype (with parameters)	Comments
_chdir(path as zstring ptr) as integer	Change current directory to given path.
_getcwd(path as zstring ptr, numchars as integer) as zstring ptr	Returns name of current working directory.
_mkdir(path as zstring ptr) as integer	Create a directory using given path name.
_rmdir(path as zstring ptr) as integer	Delete a specified directory.

File Manipulation

```
#include "crt/sys/stat.bi"
#include "crt/io.bi"
```

Prototype (with parameters)	Comments
chmod(path as zstring ptr, pmode as integer) as integer	Change permission settings of a file.

fstat(handle as integer, buffer as type stat ptr) as integer	Get file status information.
remove(path as zstring ptr) as integer	Delete a named file.
rename_(oldname as zstring ptr, newname as zstring ptr) as integer	rename a file.
stat(path as zstring ptr, buffer as type stat ptr) as integer	Get file status information of named file.
umask(pmode as uinteger) as uinteger	Set file permission mask.

Stream I/O

#include "crt/stdio.bi"

Prototype (with parameters)	Comments
clearerr(file_pointer as FILE ptr)	Clear error indicator of stream,
fclose(file_pointer as FILE ptr) as integer	Close a file,
feof(file_pointer as FILE ptr) as integer	Check if end of file occurred on a stream.
ferror(file_pointer as FILE ptr) as integer	Check if any error occurred during file I/O.
fflush(file_pointer as FILE ptr) as integer	Write out (flush) buffer to file.
fgetc(file_pointer as FILE ptr) as integer	Get a character from a stream.
fgetpos(file_pointer as FILE ptr, fpos_t current_pos) as integer	Get the current position in a stream.
fgets(string1 as zstring ptr, maxchar as integer, file_pointer as FILE ptr) as zstring ptr	Read a zstring from a file.
fopen(filename as zstring ptr, access_mode as zstring ptr) as FILE ptr	Open a file for buffered I/O.
fprintf(file_pointer as FILE ptr, format_string as zstring ptr, args) as integer	Write formatted output to a file,
fputc(c as integer, file_pointer as FILE ptr) as integer	Write a character to a stream.
fputchar(c as integer) as integer	Write a character to stdout.
fputs(string1 as zstring ptr, file_pointer as FILE ptr) as integer	Write a zstring to a stream.
fread(buffer as zstring ptr, size as size_t count as size_t, file_pointer as FILE ptr) as size_t	Read unformatted data from a stream into a buffer.
freopen(filename as zstring ptr, access as zstring ptr mode, file_pointer as FILE ptr) as FILE ptr	Reassign a file pointer to a different file.

fscanf(file_pointer as FILE ptr, format as zstring ptr, zstring, args) as integer	Read formatted input from a stream.
fseek(file_pointer as FILE ptr, offset as long, origin as integer) as integer	Set current position in file to a new location.
fsetpos(file_pointer as FILE ptr, current_pos as fpos_t) as integer	Set current position in file to a new location.
ftell(file_pointer as FILE ptr) as long	Get current location in file.
fwrite(buffer as zstring ptr, size as size_t, count as size_t, file_pointer as FILE ptr) as size_t	Write unformatted data from a buffer to a stream.
getc(file_pointer as FILE ptr) as integer	Read a character from a stream.
getchar() as integer	Read a character from stdin.
gets(buffer as zstring ptr) as zstring ptr	Read a line from stdin into a buffer.
printf(format as zstring ptr, _string, args) as integer	Write formatted output to stdout.
putc(c as integer, file_pointer as FILE ptr) as integer	Write a character to a stream.
putchar(c as integer) as integer	Write a character to stdout.
puts(string1 as zstring ptr) as integer	Write a zstring to stdout.
rewind(file_pointer as FILE ptr)	Rewind a file.
scanf(format_string as zstring ptr, args) as integer	Read formatted input from stdin.
setbuf(file_pointer as FILE ptr, buffer as zstring ptr)	Set up a new buffer for the stream.
setvbuf(file_pointer as FILE ptr, buffer as zstring ptr, buf_type as integer, buf as size_t size) as integer	Set up new buffer and control the level of buffering on a stream.
sprintf(string1 as zstring ptr, format_string as zstring ptr, args) as integer	Write formatted output to a zstring.
sscanf(buffer as zstring ptr, format_string as zstring ptr, args) as integer	Read formatted input from a zstring.
tmpfile() as FILE ptr	Open a temporary file.
tmpnam(file_name as zstring ptr) as zstring ptr	Get temporary file name.
ungetc(c as integer, file_pointer as FILE ptr) as integer	Push back character into stream's buffer

Low level I/O

```
#include "crt/io.bi"
```

So far Win32 only, connects to MSVCRT.DLL (headers missing for

other platforms)

Prototype (with parameters)	Comments
<code>_close(handle as integer) as integer</code>	Close a file opened for unbuffered I/O.
<code>_creat(filename as zstring ptr, pmode as integer) as integer</code>	Create a new file with specified permission setting.
<code>_eof(handle as integer) as integer</code>	Check for end of file.
<code>_lseek(handle as integer, offset as long, origin as integer) as long</code>	Go to a specific position in a file.
<code>_open(filename as zstring ptr, oflag as integer, pmode as uinteger) as integer</code>	Open a file for low-level I/O.
<code>_read(handle as integer, buffer as zstring ptr, length as uinteger) as integer</code>	Read binary data from a file into a buffer.
<code>_write(handle as integer, buffer as zstring ptr, count as uinteger) as integer</code>	Write binary data from a buffer to a file.

Mathematics

`#include "crt/math.bi"`

Prototype (with parameters)	Comments
<code>abs_(n as integer) as integer</code>	Get absolute value of an integer.
<code>acos_(x as double) as double</code>	Compute arc cosine of x.
<code>asin_(x as double) as double</code>	Compute arc sine of x.
<code>atan_(x as double) as double</code>	Compute arc tangent of x.
<code>atan2_(y as double, x as double) as double</code>	Compute arc tangent of y/x.
<code>ceil(x as double) as double</code>	Get smallest integral value that exceeds x.
<code>cos_(x as double) as double</code>	Compute cosine of angle in radians.
<code>cosh(x as double) as double</code>	Compute the hyperbolic cosine of x.
<code>div(number as integer, denom as integer) as div_t</code>	Divide one integer by another.
<code>exp_(x as double) as double</code>	Compute exponential of x.
<code>fabs(x as double) as double</code>	Compute absolute value of x.
<code>floor(x as double) as double</code>	Get largest integral value less than x.

fmod(x as double, y as double) as double	Divide x by y with integral quotient and return remainder.
frexp(x as double, expptr as integer ptr) as double	Breaks down x into mantissa and exponent of no.
labs(n as long) as long	Find absolute value of long integer n.
ldexp(x as double, exp as integer) as double	Reconstructs x out of mantissa and exponent of two.
ldiv(number as long, denom as long) as ldiv_t	Divide one long integer by another.
log_(x as double) as double	Compute log(x).
log10(x as double) as double	Compute log to the base 10 of x.
modf(x as double, intptr as double ptr) as double	Breaks x into fractional and integer parts.
pow(x as double, y as double) as double	Compute x raised to the power y.
rand() as integer	Get a random integer between 0 and 32767.
random(max_num as integer) as integer	Get a random integer between 0 and max_num.
randomize()	Set a random seed for the random number generator.
sin_(x as double) as double	Compute sine of angle in radians.
sinh(x as double) as double	Compute the hyperbolic sine of x.
sqrt(x as double) as double	Compute the square root of x.
srand(seed as uinteger)	Set a new seed for the random number generator (rand).
tan_(x as double) as double	Compute tangent of angle in radians.
tanh(x as double) as double	Compute the hyperbolic tangent of x.

Memory Allocation

#include "crt/stdlib.bi"

Prototype (with parameters)	Comments
calloc(num as size_t elems, elem_size as size_t) as any ptr	Allocate an array and initialise all elements to zero .
free(mem_address as any ptr)	Free a block of memory.
malloc(num as size_t bytes) as any ptr	Allocate a block of memory.
realloc(mem_address as any ptr, newsize as	Reallocate (adjust size) a block of

size_t) as any ptr	memory.
--------------------	---------

Process Control

```
#include "crt/stdlib.bi"
```

Prototype (with parameters)	Comments
abort()	Abort a process.
execl(path as zstring ptr, arg0 as zstring ptr, arg1 as zstring ptr, ..., NULL) as integer	Launch a child process (pass command line).
execlp(path as zstring ptr, arg0 as zstring ptr, arg1 as zstring ptr, ..., NULL) as integer	Launch child (use PATH, pass command line).
execv(path as zstring ptr, argv as zstring ptr) as integer	Launch child (pass argument vector).
execvp(path as zstring ptr, argv as zstring ptr) as integer	Launch child (use PATH, pass argument vector).
exit_(status as integer)	Terminate process after flushing all buffers.
getenv(varname as zstring ptr) as zstring ptr	Get definition of environment variable,
perror(string1 as zstring ptr)	Print error message corresponding to last system error.
putenv(envstring as zstring ptr) as integer	Insert new definition into environment table.
raise(signum as integer) as integer	Generate a C signal (exception).
system_(string1 as zstring ptr) as integer	Execute a resident operating system command.

Searching and Sorting

```
#include "crt/stdlib.bi"
```

Note: The *compare* callback function required by `bsearch` and `qsort` must be declared as `cdec1`. It must return a value `<0` if its first argument should be located before the second one in the sorted array `>0` if the first argument should be located after the second one, and

zero if their relative positions are indifferent (equal values).

Prototype (with parameters)	Comments
bsearch(key as any ptr, base as any ptr, num as size_t, width as size_t, compare as function(elem1 as any ptr, elem2 as any ptr) as integer) as any ptr	Perform binary search.
qsort(base as any ptr, num as size_t, width as size_t, compare as function(elem1 as any ptr, elem2 as any ptr) as integer)	Use the quicksort algorithm to sort an array.

String Manipulation

```
#include "crt/string.bi"
```

Prototype (with parameters)	Comments
strcpy(dest as zstring ptr, src as zstring ptr) as zstring ptr	Copy one zstring into another.
strcmp(string1 as zstring ptr, string2 as zstring ptr) as integer	Compare string1 and string2 to determine alphabetic order.
strcpy(string1 as zstring ptr, string2 as zstring ptr) as zstring ptr	Copy string2 to string1.
strerror(errno as integer) as zstring ptr	Get error message corresponding to specified error number.
strlen(string1 as zstring ptr) as integer	Determine the length of a zstring.
strncat(string1 as zstring ptr, string2 as zstring ptr, n as size_t) as zstring ptr	Append n characters from string2 to string1.
strncmp(string1 as zstring ptr, string2 as zstring ptr, n as size_t) as integer	Compare first n characters of two strings.
strncpy(string1 as zstring ptr, string2 as zstring ptr, n as size_t) as zstring ptr	Copy first n characters of string2 to string1.
strnset(string1 as zstring ptr, c as integer, size_t n) as zstring ptr	Set first n characters of zstring to c.
strrchr(string1 as zstring ptr, c as integer) as zstring ptr	Find last occurrence of character c in zstring.

Time

#include "crt/time.bi"

Prototype (with parameters)	Comments
asctime(time as type tm ptr) as zstring ptr	Convert time from type tm to zstring.
clock() as clock_t	Get elapsed processor time in clock ticks.
ctime(time as time_t ptr) as zstring ptr	Convert binary time to zstring.
difftime(time_t time2, time_t time1) as double	Compute the difference between two times in seconds.
gmtime(time as time_t ptr) as type tm ptr	Get Greenwich Mean Time (GMT) in a tm structure.
localtime(time as time_t ptr) as type tm ptr	Get the local time in a tm structure.
time_(timeptr as time_t ptr) as time_t	Get current time as seconds elapsed since 0 hours GMT 1/1/70.

See also

- **#include**

File I/O with FreeBASIC



In FreeBASIC, there are 4 possible ways to perform file I/O:

1. Using the built-in BASIC commands like **open** , **get** , **put** , and **close**. These are the most portable and easiest to use. They work on all platforms supported by FreeBASIC. Open files are identified by "file number" which can't be passed into functions from below.
2. Using the C stream I/O functions like `fopen`, `fread`, `ftell`, `fclose` (see `Stream` of the C library FreeBASIC relies on. This is slightly faster than and adds a bit of overhead but is still well portable. Open files are identified by file pointers, as in the C language. The `FileAttr` function can be used to return a stream I/O pointer from a file number.
3. Using the C low-level I/O functions like `_open`, `_read`, `_write`, `_close` (**Functions**). Those functions should be portable, but so far headers are not available for all platforms. They will not compile to any other platform by now.
4. Talk directly to the OS kernel (DOS: use DOS and DPMI INT's , Win32: use `kernel32`). This is no longer portable. Files are identified by handles generated by `kernel32`.

This example shows and compares methods 1. and 2. described above, using the `FileAttr` and `fread` functions used. It expects 2 commandline arguments, providing names of files to be read and compared. It compares the reading performance (make sure the file cache is empty before running).

Example

```
Data " File I/O example & test GET vs FREAD | (CL)
Data " http://www.freebasic.net/wiki/wikka.php?wak
Rem
Rem Compile With FB 0.20 Or newer
Rem
Rem In the commandline supply preferably 2 different files
Rem Default Is "BLAH" For both (bad)
Rem In both loops (Get And FREAD) the last Read call is the slowest

#include "crt\stdio.bi" ' Otherwise the "C"-stuff is not available

Dim As FILE Ptr QQ ' This is the C-like file pointer
```

```

Dim As UByte Ptr    BUF  '' Buffer used for both FE
Dim As UInteger     FILN '' FB-like "filename"

Dim As UInteger     AA, BB, CC, DD, EE
Dim As ULongInt     II64 '' We do try to support fi

Dim As String       VGSTEMP, VGSDFILE1, VGSDFILE2

? : Read VGSTEMP : ? VGSTEMP : Read VGSTEMP : ? VG

VGSTEMP=Command$(1) : VGSDFILE1="BLAH"
If (VGSTEMP<>"") Then VGSDFILE1=VGSTEMP
VGSTEMP=Command$(2) : VGSDFILE2=VGSDFILE1
If (VGSTEMP<>"") Then VGSDFILE2=VGSTEMP

BUF = Allocate(32768) '' 32 KiB - hoping it won't

? : ? "FB - OPEN - GET , """+VGSDFILE1+""": Sleep
FILN = FreeFile : AA=0 : II64=0 '' AA counts block
BB=Open (VGSDFILE1 For Binary Access Read As #FILN)
'' Result 0 is OK here, <>0 is evil
'' "ACCESS READ" should prevent file creation if i
? "OPEN result : " ; BB
If (BB=0) Then '' BB will be "reused" for timer be
BB=Cast(UInteger,(Timer*100)) '' No UINTEGER TIM
CC=Get (#FILN,, *BUF,32768,DD)
'' CC has the success status, 0 is OK, <>0 is ba
'' DD is the amount of data read
'' EOF is __NOT__ considered as error here
? "0th GET      : ";CC;" ";DD
? "2 bytes read : ";BUF[0];" ";BUF[1]
Do
  AA=AA+1 : II64=II64+Cast(ULongInt,DD)
  If (DD<32768) Or (CC<>0) Then Exit Do '' Give
  CC=Get (#FILN,, *BUF,32768,DD)
Loop
EE=Cast(UInteger,(Timer*100))-BB
? "Time          : ";(EE+1)*10;" ms"
If (AA>1) Then ? "Last GET      : ";CC;" ";DD

```

```

? "Got __EXACTLY__ ";II64;" bytes in ";AA;" call
Close #FILN
ENDIF

? : ? "C - FOPEN - FREAD , """+VGSDFILE2+"""" : Sle
AA=0 : II64=0 '' AA counts blocks per 32 KiB alrea
QQ=FOPEN(VGSDFILE2,"rb")
'' Here 0 is evil and <>0 good, opposite from above
'' File will not be created if it doesn't exist (g
'' "rb" is case sensitive and must be lowercase, S
? "FOPEN result : " ; QQ
If (QQ<>0) Then
  BB=Cast(UInteger,(Timer*100)) '' No UINTEGER TIM
  DD=FREAD(BUF,1,32768,QQ) '' 1 is size of byte -
  '' Returns size of data read, <32768 on EOF, 0 a
  ? "0th FREAD : ";DD
  ? "2 bytes read : ";BUF[0];" ";BUF[1]
  Do
    AA=AA+1
    If (DD<=32768) Then II64=II64+Cast(ULongInt,DD)
    If (DD<>32768) Then Exit Do '' ERR or EOF
    DD=FREAD(BUF,1,32768,QQ)
  Loop
  EE=Cast(UInteger,(Timer*100))-BB
  ? "Time : ";(EE+1)*10;" ms"
  If (AA>1) Then ? "Last FREAD : ";DD
  ? "Got __EXACTLY__ ";II64;" bytes in ";AA;" call
  FCLOSE(QQ)
ENDIF

Deallocate(BUF): Sleep 1000 '' Crucial

End

```

See also

- **File I/O Functions**
- **C Standard Library Functions**
- **Get (File I/O Command)**



freebasic
community tutorials

Tutorials submitted by the FreeBASIC community:

Getting Started

- **Getting Started with FreeBASIC** by *SJ Zero*
- **Using libraries in FreeBASIC** by *SJ Zero*
- **Using the Mouse in FreeBASIC** by *MystikShadows*
- **Get Information into your program** by *TekRat*
- **Using Dynamic Arrays in FreeBASIC** by *SephKnows*
- **Beginners Guide to Types as Objects (Part 1)** by *YetiFoot*
- **Beginners Guide to Types as Objects (Part 2)** by *YetiFoot*
- **Introduction to Variable Scope** by *rdc*
- **Introduction to Arrays** by *rdc*
- **Introduction to the Type**

Intermediate Techniques

- **Introduction to Function Overloading in FreeBASIC** by *:stylin:*

Mathematics

- **Different ways angles are measured** by *RandyKeeling*
- **A Brief Introduction To Trigonometry** by *RandyKeeling*

Windows API

- **Introduction to Message-Based Programming** by *rdc*

Libraries

- **Interfacing with C** by *UtenNavn*
- **SDL_NET** by *Paragon*
- **Using FreeBASIC Built Libraries with GCC** by

Def by rdc

- **New To Programming?**
by The FB Community
- **Compiling a BIG QB program** *by Antoni*

Game Programming

- **How to Program a Game: Lesson 1** *by Lachie Dazdarian*
- **Managing A High Score Table** *by Lachie Dazdarian*

Flow Control Statements

- **The IF Statement** *by rdc*
- **The Select Case Statement** *by rdc*

Pre Processor

- **Conditional Compilation And You** *by AetherFox*

Memory Management

- **Introduction to Pointers** *by rdc*
- **Pointers, Data Types and Memory** *by rdc*
- **The Pointer Data Type** *by rdc*
- **Using Linked Lists** *by Parker*
- **Dynamic Arrays in Types** *by rdc*

Jeff Marshall

Object Oriented Programming

- **Introduction to the Extended Type** *by rdc*
- **Simulating Polymorphism** *by rdc*
- **OOP in non-OOP languages** *by KevinWhitefoot*
- **Const Qualifiers and You** *by notthecheatr*

FBgfx

- **Creating and Understanding Your FBgfx Img and Font Buffer** *by The FB Community*

FreeBASIC Code, Games, and Libraries. Written in FreeBASIC, by FreeBASIC Community Members.

Code Editors & IDEs

FBEdit, an IDE for FB by KetilO (Win32)

JellyFish Pro, an IDE for FB by Paul Squires (Win32)

VISG GUI Builder (WIN) by mrhx

EzeeGui GUI builder (WIN) by Jerry Fielden

Graphics Code

Demos

The FreeBASIC GFX Demo

Central by Adigun A.Polack

Animated Clouds by Zamaster

Island Generation by rdc

Plasma Generation by

Zamaster

Graphics Functions and Primitives

AntiAliased Bezier Curves by Acetoline

Antialiased Circles by Acetoline

Ellipse Renderer by Pritchard

Catmull-rom Splines by relsoft

Bezier vs Catmull-rom by relsoft

Accurate Image Scaler by KristopherWindsor

Spline Curve by Zamaster

FreeBASIC Games

FreeBASIC Games Directory by Lachie Dazdarian

Featured Games:

Cute Short Game Project by redcrab

Kingdoms by Piptol

Lynn's Legacy by cha0s and Josiah Tobin

Relativity by Lithium

Star Cage by Lachie Dazdarian

100 Line Tetris by Deleter

Any PNG or JPEG as a Jigsaw Puzzle by Mysoft

GUI Code

In Game GUI by coderJeff

Zine GUI by VonGodric

WX GUI example by ciw1973

KwikGUI (WIN/LIN/DOS) by

Vincent DeCampo

FB_GUI by BasicScience

Networking- Web Code

FB Web Server (Win) by parakeet

FB Server side scripting (uses the server above) by fishhf

ChiSock portable sockets library by cha0s

Rotozoom by Dr_D

Colors and Palettes

24bit to 16bit color width by

Eternal_Pain

HSV Color Space by Antoni

Formats

fbpng library by yetifoot

JPEG image loader by Antoni

3D

Tree Generation by Zamaster

Quadtree-Based Renderer by

relsoft

Animation

ASCII Animation Example by

Pritchard

Chain-Like Animation Tutorial

by Lachie Dazdarian

Sound Code

Mic Input using FMod by

mambazo

Using the PC Speaker by

several

Wave synthesizer by Zamaster

Math Code

FBMath by jdebord

A* Pathfinding by dumbledore

Fraction Library by Zamaster

Big Number Wrapper by

Yetifoot

BCD arithmetics by srvaldez

10Byte extended float by

srvaldez, included in FB

examples

I/O Code

Text Input by Pritchard &

sir_mud

ConLib Console library with

PCopy by cha0s

Lock Mouse to Grid Positions

by Pritchard

Serial Port

Drive a Parallax servo

controller by phishguy

Modbus device finder by

Antoni

Serial port terminal program by

Antoni

OS Specific Code

Windows

ServiceFB (Win) by zerospeed

FBWinPrint 1.0 by vdecampo

In memory dialogs by

MichaelW

Talking program usin Win

Voice API, by coder guy

Using GfxLib in Windows API

by MichaelW

Print a bitmap file by MichaelW

ShellExecute wrapper by

RayBritton

FBWiki to chm format

converter by coderjeff

FB ODBC library by KaraK

Get a file from an URL by

Sisophon

Linux

Printing on Linux by coderJeff

Using GfxLib on Gtk by caseih

**CRC Calculation by
Fragmeister**

Physics simulation

**Atom smash simulation by
coderjeff**

**2d rigid body library by
coderjeff**

Irrlicht wrapper + Newton

Intergrated by SiskinEDGE

Text/Parser Code

**Cross Platform INI library by
SirMud**

Expression Parser by yetifoot

Turing Machine by Zamaster

Roman Numeral to Integer

Conversion by stylin

**Unicode console calender, by
zippy and voodooattack**

**FB source to highlighted HTML
by Kristopher Windsor**

**Portable help (not .chm) viewer
by coderjeff**

Lisp interpreter by coderjeff

Cryptography

MARS encryption by Zamaster

**AES Encryption/Decryption by
Zamaster**

DES/LUCIFER

**Encryption/Decryption by
Zamaster**

MD5 Calculator by DOS386

Tiger Hash by Mindless

DOS

**Detect system codepage by
DrV**

**Calling an Interrupt requiring a
pointer by DrV**

**Access BTRIEVE files by mjs
"GetDiskFreeSpaceEx" Check**

**for disk total/free space on
FAT32 by DOS386**

DPMI host detection

version/capabilities by DrV

Data structures and special- purpose UDTs

Boolean Type by Imortis

Safe FBstring Type by stylin

FreeBASIC Memory Leak

Detector by DrV & Others

Auto-deallocating 'Smart'

Pointers by stylin

UDTs for Properties by

Pritchard

Miscellaneous Code

FreeBASIC Extended Library

FB CAD by owen

**FBstd C++ Lib Port (W.I.P.) by
stylin**

Testly by zerospeed

**Portable way to add a resource
to a program by voodooattack**

CPU Identification by MichaelW

**Cpu Cycle counter for
benchmarking of code by**

MichaelW

**Use of the FBGfx built-in LZW
routines by Lillo**

**Using FB dll's in RapidQ
programs by JohnK**

Community Websites/Links

External Library

Documentation

Sourceforge

FreeBASIC Games Directory

This is a place to post worthy projects/code snippets for FreeBASIC, in their relative categories. To add a page, link to either its wiki page, website, or thread on the FreeBASIC Forums. State the project name and who it's by. Sections may be broken down into their own separate pages some time in the future. Note: Due to FB being in Beta stage of development, earlier coded projects may need to be reconfigured or recompiled to work on later versions of FreeBASIC.

External Libraries Index



This is the list of external library bindings currently included in FreeBASIC. Visit the link shown for each individual library below to see more information. To obtain a needed external library or DLL, please visit the library's homepage. If you translated additional headers, or updated existing ones, please post them on the [FreeBASIC forum](#) or submit them to the [fbc project's patch tracker](#)!

Graphical/test-based user interfaces

CGUI - Library for making GUIs in a simple way.

Curses - Standardized console user interface library.

GTK+ - Cross-platform Graphical User Interface Library.

IUP - Portable toolkit for building graphical user interfaces.

wxC - Cross-platform Graphical User Interface Library.

Windows API - Windows GUIs and more

X11 - Windowing system commonly used on Linux systems

Graphics

Allegro - Game programming library.

DUGL - Game and graphics library for DOS.

caca - A colour ASCII art library.

Cairo - 2D graphics library with support for multiple output devices.

DISLIN - Library of subroutines and functions that display data graphically.

freelut - A free alternative to **GLUT**, an OpenGL library for window creation and callback-based input handling

FreeImage - Open Source library to support popular graphics image formats.

Freetype2 - A Free, High-Quality, and Portable Font Engine.

GD - Open source code library for the dynamic creation of images by programmers.

GIFLIB - Portable tools and library routines for working with GIF images

GLUT - the original (but now inactive) OpenGL Utility Toolkit

GLFW - An OpenGL library for creating an OpenGL window and handling input from the user's main loop

GRX - 2D graphics library.

IL (DevIL) - A full featured cross-platform image library.

japi - Open source free software GUI toolkit using Java's AWT Toolkit.

jpeglib - Cross-platform library for reading/writing jpeg images.

JPGalleg - A small add-on for Allegro that adds JPG images handling capabilities to the library.

libpng - Allows reading and writing PNG images.

OpenGL - Cross-platform 3D Graphics library.

PDFlib - Portable library for dynamically generating PDF documents.

SDL - Cross-platform multimedia library.

TinyPTC - A small and easy to use framebuffer graphics library.

Music/Sound, Audio/Video

BASS - Audio library for use in Windows with a beta for Linux.

BASSMOD - BASSMOD is a MOD only (XM, IT, S3M, MOD, MTM, UM) version of BASS

Flite - Run time speech synthesis engine

FMOD - Audio library supporting just about any format.

MediaInfo - Library to read out technical and tag information from many media file formats

mpg123 - MPEG (including MP3) decoder library

Ogg - Ogg multimedia container format creation/decoder library

OpenAL - Cross-platform 3D audio API.

PortAudio - Cross-platform audio I/O library

sndfile - Library to read/write/convert audio files in various formats

VLC - Audio/video playback

Vorbis - Ogg Vorbis audio compression library

Database

GDBM - Database functions using extensible hashing, primarily for storing key/data pairs to data files

MySQL - High-Quality, widely used database engine.

PostgreSQL - Free software object-relational database management system

SQLite - Small C library that implements a self-contained, embeddable, zero-configuration SQL database engine.

Development Helpers

CUnit - Lightweight system for writing, administering, and running unit tests in C.

GDSL - The Generic Data Structures Library is a collection of routines for

generic data structures.

gettext (includes libintl) - Internationalization mechanism

GNU ASpell - Free and Open Source spell checker.

libbfd - Allows applications to use the same routines to operate on object files whatever the object file format.

Embeddable Languages

JNI - Standard programming interface for writing Java native methods and embedding the Java virtual machine into native applications.

json-c - A JSON implementation in C

libffi - Foreign function interface and closure library

libjit - Runtime (just in time) compilation library

Lua - Lightweight, embeddable scripting engine using the Lua language

SpiderMonkey - Embeddable javascript engine.

Cryptography

cryptlib - A powerful security toolkit which allows even inexperienced crypto programmers to easily add encryption and authentication services to their software.

UUID - Universally Unique Identifier generation and parsing library

Mathematics

big_int - Library for using arbitrarily large integers.

Chipmunk - 2D rigid body physics library

GMP - Free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating point numbers.

GSL - Provides a wide range of mathematical routines such as random number generators, special functions and least-squares fitting.

Newton - Integrated solution for real time simulation of physics environments.

ODE - Open source, high performance library for simulating rigid body dynamics.

Networking

cgi-util - Small C library for creating CGI programs for Websites.

curl - Free and easy-to-use client-side URL transfer library supporting almost every protocol.

FastCGI - Open extension to CGI that provides high performance without the limitations of server specific APIs.

ZeroMQ - High-performance asynchronous messaging library

eXtensible Markup Language (XML)

Expat - Stream oriented XML parser library with several useful features.

libxml - De-facto standard library for accessing xml files.

libxslt - XSLT itself is a an XML language to define transformation for XML.

Mini-XML - Small XML parsing library that you can use to read XML and XML-like data files in your application.

Regular Expressions

PCRE - Regular expression pattern matching using the same syntax as Perl.

TRE - Lightweight, robust, and efficient POSIX compliant regexp matching library.

Compression

bzip2 - For reading/writing .bz2 files or in-memory (de)compression using the bzip2 algorithms

libzip - Easy-to-use library for creating and unpacking .zip files

liblzma - Strong LZMA-based compression library used for .lzma and .xz file formats

LZO - Offers fast compression and very fast decompression.

QuickLZ - Very fast Compression Library

zlib - Lossless data compression library unencumbered by patents.

System APIs

C Runtime Library

DOS API

disphelper - Helper library to use COM objects from plain C

GLib - GNOME's universal cross-platform software utility library

Windows API

X11 - Windowing system commonly used on Linux systems

User Contributed Libraries

Visit http://www.freebasic.net/___old_site/arch/ for other libraries.

Library for making GUIs in a simple way.

Website: <http://cgui.sourceforge.net/index.html>,
<http://www.allegro.cc/resource/Libraries/GUI/CGUI>

Platforms supported: Win32, Linux

Headers to include: `cgui.bi`

Header version: 2.0.4

Example Usage: yes, in `examples/GUI/CGUI/`

Standardized console user interface library

Website: <http://pdcurses.sourceforge.net/> and <http://www.gnu.org/software>

Platforms supported: DOS, Win32, Linux

Headers to include: curses.bi

Header versions: pdcurses 3.4, ncurses 5.9

Note: On Win32 systems pdcurses is used, on Linux it uses the standard

Examples: yes, in examples/console/curses/

Example

```
#include once "curses.bi"

initscr()
cbreak()
noecho()
start_color()

'' The default pair 0 will have the console's default colors

'' Set pair 1 to be white/blue
init_pair(1, COLOR_WHITE, COLOR_BLUE)

'' Select pair 1, so from now on output will be white on blue
attrset(COLOR_PAIR(1))

printw(!"Hello, world!\n")

'' Reset to pair 0
attrset(COLOR_PAIR(0))

'' Sleep
printw(!"Waiting for keypress...\n")
getch()

endwin()
```


GTK+, The GIMP ToolKit



Cross-platform Graphical User Interface library

Website: <http://www.gtk.org>

Platforms supported: Win32, Linux

Headers to include: gtk/gtk.bi

Example Usage: yes, in examples/GUI/GTK+/
Header version: 2.24.27, 3.14.10

By default, gtk/gtk.bi will use the GTK+ 2 API.

Define `__USE_GTK3__` before including gtk/gtk.bi to use GTK+ 3.

Example

```
#include once "gtk/gtk.bi"

Dim Shared As GtkWidget Ptr win

Private Sub on_clicked cdecl(ByVal button As GtkBu
    Static As Integer clickcount = 0
    clickcount += 1
    gtk_window_set_title(GTK_WINDOW(win), "clicked
End Sub

gtk_init(NULL, NULL)

win = gtk_window_new(GTK_WINDOW_TOPLEVEL)
gtk_window_set_title(GTK_WINDOW(win), "A small GTK
gtk_window_set_default_size(GTK_WINDOW(win), 300,
gtk_container_set_border_width(GTK_CONTAINER(win),

g_signal_connect(G_OBJECT(win), "destroy", G_CALLE

Dim As GtkWidget Ptr button = gtk_button_new_with_
gtk_container_add(GTK_CONTAINER(win), button)

g_signal_connect(G_OBJECT(button), "clicked", G_CA
```

```
gtk_widget_show_all(win)
```

```
gtk_main()
```

Portable toolkit for building graphical user interfaces.

Website: <http://www.tecgraf.puc-rio.br/iup/>

Platforms supported: Win32, Linux

Headers to include: IUP/iup.bi

Header version: 3.13

Example Usage: yes, in examples/GUI/IUP/

wx-c, C Interface for WxWidgets



Cross-platform Graphical User Interface library

Website: <http://wxnet.sourceforge.net/>

Platforms supported: Win32, Linux

Headers to include: wx-c/wx.bi

Header version: 0.9.0.2

Example Usage: yes, in examples/GUI/wx-c/

Windows API



Standard API for all Windows Systems, used for example for creating Windows GUIs (forms and controls), socket programming, inter-process communication, and so much more.

Website: <http://msdn.microsoft.com/en-us/library/ee663300.aspx>

Platforms supported: Win32, Linux (using WINE)

Headers to include: windows.bi

Examples: yes, in examples/win32/

The X Windowing System is widely used on Linux as the layer that coordinates drawing to the screen by providing windows. It also delivers events such as mouse and keyboard input from the kernel to applications. It is designed to run as a server that can be contacted through a specific protocol. The client's side of the protocol is implemented by libraries such as the old Xlib or the more modern XCB. Applications can use these to create windows and draw to them. However, typically most developers will choose to use a GUI library such as GTK+ (which has an X11 backend) instead.

Website: <http://www.x.org/>, <http://xcb.freedesktop.org/>

Platforms supported: Linux

Headers to include: X11/*.h

Allegro



Game programming library

Website: <http://alleg.sourceforge.net/index.html>

Platforms supported: Win32, Linux, DOS

Headers to include: allegro.bi (Allegro 4) or allegro5/allegro.bi (Allegro 5)

Header version: 4.4.2, 5.0.11

Example Usage: yes, in examples/graphics/Allegro/

Graphics and game programming library

Website: <http://dugl.50webs.com>

Platforms supported: DOS

Headers to include: DUGL.BI (not yet included with FB, see link below)

Example of usage: see link below

Note: use DUGL 1.13 or newer (see link below), older version have a bug and do crash when used with FB

See forum thread: <http://www.freebasic.net/forum/viewtopic.php?t=11046>

A colour ASCII art library.

Website: <http://libcaca.zoy.org/>

Platforms supported: Win32, Linux, DOS

Headers to include: caca.bi (new API) or caca0.bi (old API)

Header version: libcaca-0.99.beta19

Example Usage: yes, in examples/console/caca/

2D graphics library with support for multiple output devices. It can be used in a Win32 window or device context.

Website: <http://www.cairographics.org>

Platforms supported: Win32, Linux

Headers to include: `cairo/cairo.bi`

Header version: 1.14.2

Examples: yes, in `examples/graphics/cairo/`

Example

```
' ' Example showing cairo being used to draw into t
#include once "cairo/cairo.bi"

Const SCREEN_W = 400
Const SCREEN_H = 300

ScreenRes SCREEN_W, SCREEN_H, 32

' ' Create a cairo drawing context, using the FB sc
Var surface = cairo_image_surface_create_for_data(

Var c = cairo_create(surface)

ScreenLock()

' ' Draw the entire context white.
cairo_set_source_rgba(c, 1, 1, 1, 1)
cairo_paint(c)

' ' Draw a red line
cairo_set_line_width(c, 1)
cairo_set_source_rgba(c, 1, 0, 0, 1)
cairo_move_to(c, 0, 0)
cairo_line_to(c, SCREEN_W - 1, SCREEN_H - 1)
cairo_stroke(c)
```

```
ScreenUnlock()
```

```
Sleep
```

```
'' Clean up the cairo context  
cairo_destroy(c)
```

Library of subroutines and functions that display data graphically.

Website: <http://www.mps.mpg.de/dislin/>

Platforms supported: Win32, Linux

Headers to include: `dislin.bi`

Header version: from 2005

Free alternative to the OpenGL Utility Toolkit



Just like **GLUT**, freeglut is a helper library that can be used to create OpenGL applications. It allows easy creation of windows with OpenGL drawing contexts and callback-based input event handling.

Website: <http://freeglut.sourceforge.net/>

Platforms supported: Win32, Linux

Headers to include: GL/freeglut.bi

Header version: 3.0.0

FreeImage is an Open Source library project for developers who would like PNG, BMP, JPEG, TIFF and others as needed by today's multimedia (multithreading safe, compatible with all 32-bit versions of Windows, and OS X).

Website: <http://freeimage.sourceforge.net/>

Platforms supported: Win32, Linux

Headers to include: FreeImage.bi

Header version: 3.15.1

Example included: yes, in examples/files/FreeImage

Example

Here follows an example of using FreeImage in FreeBASIC. If using V available from the [FreeImage site](#).

```
' ' Code example for loading all common image types
' ' The example loads an image passed as a command

' ' The function FI_Load returns a null pointer (0)
' ' loading. Otherwise it returns a 32-bit PUT com

#include "FreeImage.bi"
#include "crt.bi"
#include "fbgfx.bi"

Function FI_Load(filename As String) As Any Ptr
    If Len(filename) = 0 Then
        Return NULL
    End If

    ' Find out the image format
    Dim As FREE_IMAGE_FORMAT form = FreeImage_GetF
    If form = FIF_UNKNOWN Then
        form = FreeImage_GetFIFFromFilename(StrPtr
    End If
```

```

'' Exit if unknown
If form = FIF_UNKNOWN Then
    Return NULL
End If

'' Always load jpegs accurately
Dim As UInteger flags = 0
If form = FIF_JPEG Then
    flags = JPEG_ACCURATE
End If

'' Load the image into memory
Dim As FIBITMAP Ptr image = FreeImage_Load(form, filename, flags)
If image = NULL Then
    '' FreeImage failed to read in the image
    Return NULL
End If

'' Flip the image so it matches FB's coordinate system
FreeImage_FlipVertical(image)

'' Convert to 32 bits per pixel
Dim As FIBITMAP Ptr image32 = FreeImage_ConvertFromDib(image, 32)

'' Get the image's size
Dim As UInteger w = FreeImage_GetWidth(image)
Dim As UInteger h = FreeImage_GetHeight(image)

'' Create an FB image of the same size
Dim As fb.Image Ptr sprite = ImageCreate(w, h)

Dim As Byte Ptr target = CPtr(Byte Ptr, sprite->pixels)
Dim As Integer target_pitch = sprite->pitch

Dim As Any Ptr source = FreeImage_GetBits(image)
Dim As Integer source_pitch = FreeImage_GetPitch(image)

'' And copy over the pixels, row by row
For y As Integer = 0 To (h - 1)

```

```

        memcpy(target + (y * target_pitch), _
              source + (y * source_pitch), _
              w * 4)
    Next

    FreeImage_Unload(image32)
    FreeImage_Unload(image)

    Return sprite
End Function

ScreenRes 640, 480, 32

Dim As String filename = Command(1)

Dim As Any Ptr image = FI_Load(filename)
If image <> 0 Then
    Put (0, 0), image
Else
    Print "Problem while loading file : " & filename
End If

Sleep

```

Freetype2



A Free, High-Quality, and Portable Font Engine

Website: <http://www.freetype.org>

Platforms supported: Win32, Linux

Headers to include: freetype2/freetype.bi

Header version: 2.5.5

Examples: yes, in examples/graphics/FreeType/

Example

```
' ' Example of rendering a char using freetype
#include "freetype2/freetype.bi"

#ifdef __FB_LINUX__
Const TTF_FONT = "/usr/share/fonts/truetype/ttf-de
#else
Const TTF_FONT = "Vera.ttf"
#endif

Dim As FT_Library library
If (FT_Init_FreeType(@library) <> 0) Then
    Print "FT_Init_FreeType() failed" : Sleep : Er
End If

' '
' ' Load a font and render an '@' character on to a
' '

Dim As FT_Face face
If (FT_New_Face(library, TTF_FONT, 0, @face) <> 0)
    Print "FT_New_Face() failed (font file '" & TT
End If

If (FT_Set_Pixel_Sizes(face, 0, 200) <> 0) Then
    Print "FT_Set_Pixel_Sizes() failed" : Sleep :
```

```

End If

If (FT_Load_Char(face, Asc("@"), FT_LOAD_DEFAULT)
    Print "FT_Load_Char() failed" : Sleep : End 1
End If

If (FT_Render_Glyph(face->glyph, FT_RENDER_MODE_NC
    Print "FT_Render_Glyph() failed" : Sleep : End
End If

''
'' Draw the rendered bitmap
''

ScreenRes 320, 200, 32

Dim As FT_Bitmap Ptr bitmap = @face->glyph->bitmap

For y As Integer = 0 To (bitmap->rows - 1)
    For x As Integer = 0 To (bitmap->Width - 1)
        Dim As Integer col = bitmap->buffer[y * bi
        PSet(x, y), RGB(col, col, col)
    Next
Next

Sleep

```

Open source code library for the dynamic creation of images by programmers.

Website: <http://www.libgd.org>

Platforms supported: Win32, Linux

Headers to include: gd.bi

Header version: 2.1.0 development version

Examples: yes, in examples/files/GD/

GIFLIB is a package of portable tools and library routines for working with GIF images

Website: <http://giflib.sourceforge.net/intro.html>

Platforms supported: Win32, Linux, DOS

Headers to include: gif_lib.bi

Header version: 4.2.1, 5.0.4 (#define __GIFLIB_VER__ to 4 or 5 if needed; default = 5)

Examples: yes, in examples/files/GIFLIB/

GLUT, the OpenGL Utility Toolkit



GLUT is a helper library that can be used to create OpenGL applications. It allows easy creation of a window with an OpenGL drawing context and also handles events such as mouse and keyboard input or timers. GLUT appears to be no longer maintained, however there is an active alternative: **freeglut**.

Website: <http://www.opengl.org/resources/libraries/glut/>

Platforms supported: Win32

Headers to include: GL/glut.bi

Header version: 3.7

Examples: in examples/graphics/OpenGL/

GLFW, an OpenGL library



GLFW is a helper library that can be used to create OpenGL applications. It allows the creation of a window with an OpenGL drawing context and input handling while still allowing the program to have its own main loop.

Website: <http://www.glfw.org/>

Platforms supported: Win32, Linux

Headers to include: GL/glfw.bi, GLFW/glfw3.bi

Header version: 2.7.9, 3.1.1

Examples: in examples/graphics/OpenGL/

2D graphics library

Website: <http://grx.gnu.de/>

Platforms supported: Win32, Linux, DOS

Headers to include: `grx/grx20.bi`

Header version: 2.4.6

Examples: in `examples/graphics/grx/`

A full featured cross-platform image library.

Website: <http://openil.sourceforge.net/>

Platforms supported: Win32, Linux

Headers to include: IL/il.bi, IL/ilu.bi, IL/ilut.bi

Header version: 1.7.8

Examples: in examples/files/DevIL/

Example

```
' ' DevIL example

#include once "IL/il.bi"

' ' Version check
If (ilGetInteger(IL_VERSION_NUM) < IL_VERSION) Then
    Print "DevIL version is different"
    End 1
End If

' ' Good practice to explicitly initialize it
ilInit()

' ' Load a bitmap
Dim As ILuint fblogo
ilGenImages(1, @fblogo)
ilBindImage(fblogo)

Print "Loading fblogo.bmp..."
ilLoadImage("fblogo.bmp")

' ' Save a copy
Print "Saving a copy, fblogo-copy.bmp..."
ilEnable(IL_FILE_OVERWRITE)
ilSaveImage("fblogo-copy.bmp")
```

```
'' Clean up  
ilDeleteImages(1, @fblogo)
```

Java Application Programming Interface



Open source free software GUI toolkit using Java's AWT Toolkit

Website: <http://www.japi.de/>

Platforms supported: Win32, Linux

Headers to include: japi.bi

Header version: from 2005

Cross-platform library for reading/writing jpeg images

Website: <http://ijg.org/>

Platforms supported: Win32, Linux, DOS

Headers to include: jpeglib.bi

Header version: 6.2, 7.0, 8.4, 9.0 (#define `__JPEGLIB_VER__` to one of 6,7,8,9 if needed; default = 9)

Example Usage: yes, in examples/files/jpeglib

JPGAlleg



JPGalleg is a small addon for Allegro that adds JPG images handling capabilities to the library

Website: <http://www.ecplusplus.com/index.php?page=projects&pid;=1>

Platforms supported: Win32, Linux

Headers to include: jpgalleg.bi

Header version: 2.5

Allows reading and writing PNG images.

Website: <http://www.libpng.org/pub/png/libpng.html>

Platforms supported: Win32, Linux, DOS

Headers to include: png.h

Header versions: 1.2.53, 1.4.16, 1.5.21, 1.6.16

Examples: in examples/files/libpng/

When `#including` `png.h`, you can `#define` `__LIBPNG_VERSION` to one of 12, 14, 15, 16 in order to select the desired libpng version. The default is the latest version.

Overriding the default allows you to match the exact libpng version on your system (interesting for Linux distros which, for example, use the libpng 1.2 series instead of the latest version).

OpenGL, The Open Graphics Language



OpenGL is a standardized and widely used cross-platform 3D graphics library.

Usually OpenGL support comes as part of the system and the graphics drivers. There are many different projects providing a library that implements the main OpenGL API, and which one is used depends on the platform and system setup. For example, on Windows, the client API is implemented in Microsoft's `opengl32.dll`, while on Linux, there is for example the free Mesa3D project, which provides a libGL implementation. It depends on the used library or system setup which way the OpenGL API does its rendering, typically it uses OpenGL hardware drivers and is hardware-accelerated, but there also is software-rendered OpenGL (e.g. standalone Mesa3D). The system's graphics hardware drivers may provide additional OpenGL extensions, access to which is again system dependant.

Besides plain OpenGL, there are several utility, helper and wrapper libraries, such as **GLUT**, **freeglut** and **GLFW**, and even FreeBASIC's built-in graphics library has an OpenGL mode, see **Screen And Fb.Gfx_Opengl**.

Websites:

OpenGL standard: <http://www.opengl.org>

Mesa3D: <http://mesa3d.org/>

Windows OpenGL: <http://msdn.microsoft.com/en-us/library/dd374278.aspx>

Platforms supported: Win32, Linux

Headers to include: `GL/gl.bi`

Header version: Mesa-3D 10.5.1, MinGW-w64 3.3.0

Examples: yes, in `examples/graphics/OpenGL/`

Portable library for dynamically generating PDF documents

Website: <http://www.pdflib.com>

Platforms supported: Win32, Linux

Headers to include: pdflib.bi

Header version: 4.0.2

Examples: in examples/files/pdflib/

SDL, the Simple DirectMedia Layer



Cross-platform multimedia library

Website: <http://www.libsdl.org>

Platforms supported: Win32, Linux

Headers to include: SDL/SDL.bi or SDL2/SDL.bi

Header version: SDL 1.2.15, SDL2 2.0.3

Examples: yes, in examples/graphics/SDL/

A small and easy to use framebuffer graphics library.

Website: <http://sourceforge.net/projects/tinyptc/>

Platforms supported: Win32, Linux, DOS

Headers to include: `tinyptc.bi`

Examples: in `examples/graphics/tinyptc/`

Audio library for use in Windows with a Beta Version for Linux.

Website: <http://www.un4seen.com/bass.html>

Platforms supported: Win32, Linux (beta)

Headers to include: bass.bi

Header version: 2.4.8

Examples: in examples/sound/BASS/

Example

```
#include once "bass.bi"

Const SOUND_FILE = "test.mod"

If (BASS_GetVersion() < MAKELONG(2,2)) Then
    Print "BASS version 2.2 or above required!"
End 1
End If

If (BASS_Init(-1, 44100, 0, 0, 0) = 0) Then
    Print "Could not initialize BASS"
End 1
End If

Dim As HMUSIC test = BASS_MusicLoad(FALSE, @SOUND_
If (test = 0) Then
    Print "BASS could not load '" & SOUND_FILE & "
    BASS_Free()
End 1
End If

BASS_ChannelPlay(test, FALSE)

Print "Sound playing; waiting to keypress to stop
Sleep
```

```
BASS_ChannelStop(test)
BASS_MusicFree(test)
BASS_Stop()
BASS_Free()
```

BASSMOD is a MOD only (XM, IT, S3M, MOD, MTM, UMX) version of E else where you want to play some MOD music.

Website: <http://www.un4seen.com/bassmod.html>

Platforms supported: Win32, Linux

Headers to include: bassmod.bi

Header version: 2.0

Examples: in examples/sound/BASS/

Example

```
#include once "bassmod.bi"

Const SOUND_FILE = "test.mod"

If (BASSMOD_GetVersion() < 2) Then
    Print "BASSMOD version 2 or above required!"
End 1
End If

If (BASSMOD_Init(-1, 44100, 0) = 0) Then
    Print "Could not initialize BASSMOD"
End 1
End If

If (BASSMOD_MusicLoad(FALSE, SOUND_FILE, 0, 0, BAS
    Print "BASSMOD could not load '" & SOUND_FILE
    BASSMOD_Free()
End 1
End If

BASSMOD_MusicPlay()

Print "Sound playing; waiting for keypress to stop"
Sleep
```

```
BASSMOD_MusicStop()  
BASSMOD_MusicFree()  
BASSMOD_Free()
```

Flite is a run-time speech synthesis engine.

Website: <http://www.speech.cs.cmu.edu/flite/>

Platforms supported: Win32, Linux

Headers to include: flite/flite.bi

Header version: 1.4, machine-translated only

Audio library supporting just about any format.

Website: <http://www.fmod.org/index.php/products#FMOD3Programmers>

Platforms supported: Win32, Linux

Headers to include: fmod.bi

Header version: 3.75

Examples: in examples/sound/FMOD/

Example

```
#include once "fmod.bi"

Const SOUND_FILE = "test.mod"

If (FSOUND_GetVersion() < FMOD_VERSION) Then
    Print "FMOD version mismatch"
End 1
End If

If (FSOUND_Init(44100, 32, 0) = 0) Then
    Print "Could not initialize FMOD"
End 1
End If

Dim As FMUSIC_MODULE Ptr song = FMUSIC_LoadSong(SC
If (song = 0) Then
    Print "FMOD could not load '" & SOUND_FILE & "'
    FSOUND_Close()
End 1
End If

FMUSIC_PlaySong(song)

Print "Sound playing; waiting for keypress to stop"
Sleep
```

```
FMUSIC_FreeSong(song)
FSOUND_Close()
```

```
' ' mp3 player based on FMOD
#include once "fmod.bi"

Const SOUND_FILE = "test.mp3"

Sub print_all_tags(ByVal stream As FSOUND_STREAM Ptr)
    Dim As Integer count = 0
    FSOUND_Stream_GetNumTagFields(stream, @count)

    For i As Integer = 0 To (count - 1)
        Dim As Integer tagtype, taglen
        Dim As ZString Ptr tagname, tagvalue
        FSOUND_Stream_GetTagField(stream, i, @tagtype, @taglen, @tagname, @tagvalue)
        Print Left(*tagname, taglen)
    Next
End Sub

Function get_tag (
    ByVal stream As FSOUND_STREAM Ptr,
    ByVal tagv1 As ZString Ptr,
    ByVal tagv2 As ZString Ptr
) As String

    Dim tagname As ZString Ptr, taglen As Integer

    FSOUND_Stream_FindTagField(stream, FSOUND_TAGTYPE_UNKNOWN, @tagname, @taglen)
    If (taglen = 0) Then
        FSOUND_Stream_FindTagField(stream, FSOUND_TAGTYPE_UNKNOWN, @tagname, @taglen)
    End If

    Return Left(*tagname, taglen)
```

```
End Function
```

```
    If (FSOUND_GetVersion < FMOD_VERSION) Then  
        Print "FMOD version " + Str(FMOD_VERSION)  
    End 1
```

```
End If
```

```
    If (FSOUND_Init(44100, 4, 0) = 0) Then  
        Print "Could not initialize FMOD"  
    End 1
```

```
End If
```

```
    FSOUND_Stream_SetBufferSize(50)
```

```
    Dim As FSOUND_STREAM Ptr stream = FSOUND_Stream_Open(SOUND_FILE)
```

```
    If (stream = 0) Then
```

```
        Print "FMOD could not load '" & SOUND_FILE
```

```
        FSOUND_Close()
```

```
    End 1
```

```
End If
```

```
    ' Read out mp3 tags to show some meta information
```

```
    Print "Title:", get_tag(stream, "TITLE", "TITLE2")
```

```
    Print "Album:", get_tag(stream, "ALBUM", "TITLE2")
```

```
    Print "Artist:", get_tag(stream, "ARTIST", "TITLE2")
```

```
    'print_all_tags(stream)
```

```
    Print "Playing mp3, press a key to exit..."
```

```
    FSOUND_Stream_Play(FSOUND_FREE, stream)
```

```
    While (Inkey() = "")
```

```
        If (FSOUND_Stream_GetPosition(stream) >= FSOUND_STREAM_END)
```

```
            Exit While
```

```
        End If
```

```
        Sleep 50, 1
```

```
    Wend
```

```
    FSOUND_Stream_Stop(stream)
```

```
    FSOUND_Stream_Close(stream)
```

```
FSOUND_Close()
```

MediaInfo



MediaInfo is a cross-platform library allowing you to read out technical and tag information from audio and video files in various formats.

Website: <http://mediainfo.sourceforge.net/>

Platforms supported: Win32, Linux

Headers to include: MediaInfo.bi

Header version: from October 2011

libmpg123 is the decoder library used by the mpg123 MPEG player.

Website: <http://mpg123.org/>

Platforms supported: Win32, Linux

Headers to include: mpg123.bi

Header version: from 2010, machine-translated only

Ogg multimedia container format creation/decoder library

Website: <http://www.xiph.org/ogg/>

Platforms supported: Win32, Linux

Headers to include: ogg/ogg.bi

Header version: from 2007

OpenAL is a cross-platform 3D audio API appropriate for use with gaming applications and many other types of audio applications. ALUT is the OpenAL utility toolkit, a library providing additional functions to work with OpenAL.

Website: <http://www.openal.org>

Platforms supported: Win32, Linux

Headers to include: AL/al.bi, AL/alut.bi

Header version: openal-soft-1.16.0, freealut 1.1.0

Examples: in examples/sound/OpenAL/

PortAudio



PortAudio is a cross-platform audio I/O library that allows programs to access the system's audio devices to record or play sounds.

Website: <http://www.portaudio.com/>

Platforms supported: Win32, Linux

Headers to include: portaudio.h

Header version: from 2010, machine-translated only

libsndfile



libsndfile is a library allowing programs to access or modify audio files in various formats, for example .wav files, and also convert between them.

Website: <http://www.mega-nerd.com/libsndfile/>

Platforms supported: Win32, Linux

Headers to include: sndfile.bi

Header version: 1.0.X

Audio/video playback library from the VLC media player.

Website: <http://www.videolan.org/>, <http://wiki.videolan.org/Libvlc>

Platforms supported: Win32, Linux

Headers to include: vlc/*.bi

Header version: 1.1.x

libvorbis



Ogg Vorbis audio compression library

Website: <http://xiph.org/vorbis/>

Platforms supported: Win32, Linux

Headers to include: `vorbis/vorbisenc.bi`, `vorbis/vorbisfile.bi`

Header version: from 2007

GDBM, the GNU Database manager



Provides database functions using extensible hashing, primarily for storing key/data pairs to data files

Website: <http://www.gnu.org.ua/software/gdbm/>

Platforms supported: Win32, Linux

Headers to include: `gdbm.bi`

Header version: from 2010

MySQL



High-Quality, widely used database engine.

Website: <http://www.mysql.org>

Platforms supported: Win32, Linux

Headers to include: `mysql/mysql.bi`

Header version: 4.0.17

Examples: in `examples/database/`

PostgreSQL



Free software object-relational database management system

Website: <http://www.postgresql.org/>

Platforms supported: Win32, Linux

Headers to include: postgresql/postgres_ext.bi

Header version: from 2006

Example Usage: yes, in examples/database/

SQLite



Small C library that implements a self-contained, embeddable, zero-configuration SQL database engine.

Website: <http://sqlite.org>

Platforms supported: Win32, Linux, DOS

Headers to include: `sqlite2.bi` or `sqlite3.bi`

Header versions: 2.8.17, 3.7.8

Examples: in `examples/database/`

CUnit



Lightweight system for writing, administering, and running unit tests in C.

Website: <http://cunit.sourceforge.net/>

Platforms supported: Win32, Linux, DOS

Headers to include: CUnit/CUnit.bi

Header version: 2.1-3

Examples: in examples/misc/CUnit/

GDSL, The Generic Data Structures Library



The Generic Data Structures Library is a collection of routines for generic data structures.

Website: <http://home.gna.org/gdsl/>
Platforms supported: Win32, Linux, DOS
Headers to include: `gdsl/gdsl.bi`
Header version: from 2005
Examples: in `examples/misc/gdsl/`

An internationalization library/toolchain

Website: <http://www.gnu.org/software/gettext/gettext.html>

Platforms supported: Win32, Linux, DOS

Headers to include: libintl.bi, gettext-po.bi

Header version: from 2010, 0.17

Free and Open Source spell checker.

Website: <http://aspell.net/>

Platforms supported: Win32, Linux

Headers to include: aspell.bi

Header version: 0.60.6.1

Example

```
' ' GNU-ASspell example, converted from http://aspe
#include once "aspell.bi"

Dim As AspellConfig Ptr spell_config = new_aspell_

' ' Change this to suit the installed dictionary la
aspell_config_replace(spell_config, "lang", "en_CA

' ' Create speller object
Dim As AspellCanHaveError Ptr possible_err = new_a
If (aspell_error_number(possible_err) <> 0) Then
    Print *aspell_error_message(possible_err)
    End 1
End If
Dim As AspellSpeller Ptr speller = to_aspell_spell

Dim As String word
Do
    Print
    Input "Enter a word (blank to quit): ", word
    If (Len(word) = 0) Then
        Exit Do
    End If

    If (aspell_speller_check(speller, StrPtr(word))
        Print "Word is Correct"
```

```

Else
    Print "Suggestions:"
    Dim As AspellStringEnumeration Ptr element
        aspell_word_list_elements(aspell_spell
    Do
        Dim As ZString Ptr w = aspell_string_e
        If (w = 0) Then
            Exit Do
        End If
        Print " "; *w
    Loop
    delete_aspell_string_enumeration(elements)
End If

' - Report the replacement
'aspell_speller_store_repl(speller, misspelled
'
' - Add to session or personal dictionary
'aspell_speller_add_to_session|personal(spelle
Loop
delete_aspell_speller(speller)

```

BFD, the Binary File Descriptor Library



Provides an API to read and write object files in many different object file formats. libbfd is the core of the GNU binutils.

Website: <http://sourceware.org/binutils/>

Platforms supported: Win32, Linux, DOS

Headers to include: bfd.bi

Header version: binutils versions from 2.16 to 2.25

Define `__BFD_VER__` to 216, 217, 218, ..., 225 to include the bfd header for the corresponding binutils version.

Example

```
#define __BFD_VER__ 217
#include "bfd.bi"
```

JNI, The Java Native Interface



Standard programming interface for writing Java native methods and em

Website: <http://download.oracle.com/javase/6/docs/technotes/guides/jni/>

Platforms supported: Win32, Linux

Headers to include: jni.bi

Header version: from 2006

Examples: in examples/other-languages/Java/

Example

Three files:

- mylib.bas - A DLL writing in FreeBASIC

```
#include "jni.bi"

'' Note: The mangling must be "windows-ms" or the
Extern "windows-ms"
    Function Java_MyLib_add( env As JNIEnv Ptr, ob
        Return 1 + r
    End Function
End Extern
```

- Mylib.java - The Java class that represents the interface to the

```
(cpp)
class MyLib {
    public native int add( int l, int r );
    static {
        System.loadLibrary( "mylib" );
    }
}
```

- Test.java - The Java main() that uses the Mylib class

```
(cpp)
class Test {
    public static void main(String[] args) {
        MyLib lib = new MyLib();
        System.out.println( "2+2=" + lib.add( 2, 2 ) )
    }
}
```

Steps to test it:

- *Compile the FreeBASIC DLL:* fbc mylib.bas -dll
- *Compile the two Java classes:* javac Mylib.java Test.java
- *Run the Test class:* java Test

A JSON implementation in C

Website: <http://oss.metaparadigm.com/json-c/>

Platforms supported: Win32, Linux

Headers to include: json-c/json.bi

Header version: 0.9 (not sure)

LibFFI is a foreign function interface library allowing programs to arbitrarily take variable arguments via closures. It is used to bind native code in m

Website: <http://sourceware.org/libffi/>

Platforms supported: Windows, Linux, DOS

Headers to include: ffi.bi

Header version: 3.1

Example

Hello world:

```
#include "ffi.bi"

' Simple "puts" equivalent function
Function printer cdecl (ByVal s As ZString Ptr) As
    Print *s
    Return 42
End Function

' Initialize the argument info vectors
Dim s As ZString Ptr
Dim args(0 To 0) As ffi_type Ptr = {@ffi_type_poir
Dim values(0 To 0) As Any Ptr = {@s}

' Initialize the cif
Dim cif As ffi_cif
Dim result As ffi_status
result = ffi_prep_cif( _
    @cif, _ ' call interface object
    FFI_DEFAULT_ABI, _ ' binary interface type
    1, _ ' number of arguments
    @ffi_type_uint, _ ' return type
    @args(0) _ ' arguments
)

' Call function
```

```

Dim return_value As Integer
If result = FFI_OK Then
    s = @"Hello world"
    ffi_call(@cif, FFI_FN(@printer), @return_value

    ' values holds a pointer to the function's arg
    ' call puts() again all we need to do is chang
    ' value of s */
    s = @"This is cool!"
    ffi_call(@cif, FFI_FN(@printer), @return_value
    Print Using "Function returned &"; return_valu
End If

```

Closures:

```

#include "ffi.bi"

' Acts like puts with the file given at time of er
Sub Printer cdecl(ByVal cif As ffi_cif Ptr, ByVal
    Write #*CPtr(Integer Ptr, file), **CPtr(ZStrin
    *CPtr(UInteger Ptr, ret) = 42
End Sub

' Allocate the closure and function binding
Dim PrinterBinding As Function(ByVal s As ZString
Dim closure As ffi_closure Ptr
closure = ffi_closure_alloc(SizeOf(ffi_closure), @

If closure <> 0 Then
    ' Initialize the argument info vector
    Dim args(0 To 0) As ffi_type Ptr = {@ffi_type_

    ' Initialize the call interface
    Dim cif As ffi_cif
    Dim prep_result As ffi_status = ffi_prep_cif(
        @cif, _ ' call interface object
        FFI_DEFAULT_ABI, _ ' binary interface type
        1, _ ' number of arguments

```

```

        @ffi_type_uint, _ ' return type
        @args(0)      _ ' arguments
    )
    If prep_result = FFI_OK Then
        ' Open console file to send to PrinterBinding
        Dim ConsoleFile As Integer = FreeFile()
        Open Cons For Output As ConsoleFile

        ' Initialize the closure, setting user data
        prep_result = ffi_prep_closure_loc( _
            closure, _ ' closure object
            @cif, _ ' call interface object
            @Printer, _ ' actual closure function
            @ConsoleFile, _ ' user data, our console file
            PrinterBinding _ ' pointer to binding function
        )
        If prep_result = FFI_OK Then
            ' Call binding as a natural function call
            Dim Result As Integer
            Result = PrinterBinding("Hello World!")
            Print Using "Returned &"; Result
        End If

        Close ConsoleFile
    End If
End If

' Clean up
ffi_closure_free(closure)

```

LibJIT is a fairly straightforward, lightweight library for runtime compilation.

Website: <http://www.gnu.org/software/libjit/>

Platforms supported: Windows, Linux, DOS

Headers to include: jit.bi

Header version: git a8293e141b79c28734a3633a81a43f92f29fc2d7

Example

```
' ' Simple mul/add example

#include "jit.bi"

' initialize libjit
Dim context As jit_context_t = jit_context_create(
jit_context_build_start(context)

' define function mul_add(x, y, z)
Dim params(0 To 2) As jit_type_t = {jit_type_int,
Dim signature As jit_type_t = jit_type_create_sig(
    jit_abi_cdecl, _ ' C-style function
    jit_type_int, _ ' Return type
    @params(0), _ ' Parameter array
    3, _ ' Number of components
    1 _ ' Count references?
)
Dim mul_add As jit_function_t = jit_function_create(

' build function (result = (x*y)+z)
Dim As jit_value_t x, y, z, temp1, temp2
x = jit_value_get_param(mul_add, 0)
y = jit_value_get_param(mul_add, 1)
temp1 = jit_insn_mul(mul_add, x, y)
z = jit_value_get_param(mul_add, 2)
temp2 = jit_insn_add(mul_add, temp1, z)
jit_insn_return(mul_add, temp2)
```

```

' compile function function
jit_function_compile(mul_add)
jit_context_build_end(context)

' call function
Dim As Integer a=3, b=5, c=2, result
Dim args(0 To 2) As Integer Ptr = {@a, @b, @c}
jit_function_apply(mul_add, @args(0), @result)
Print Using "mul__add(&, &, &) = &"; a; b; c; result

' clean up libjit
jit_context_destroy(context)

```

```

'' GCD calculation example

#include "jit.bi"

' initialize libjit
Dim context As jit_context_t = jit_context_create(
jit_context_build_start(context)

' define function gcd(x as uinteger, y as uinteger)
Dim params(0 To 1) As jit_type_t = {jit_type_uint,
Dim signature As jit_type_t = jit_type_create_signature(
    jit_abi_cdecl, _ ' C-style function
    jit_type_uint, _ ' Return type
    @params(0), _ ' Parameter array
    2, _ ' Number of components
    1 _ ' Count references?
)
Dim gcd As jit_function_t = jit_function_create(context,

' build function
' check x = y
Dim As jit_value_t x, y, x_eq_y

```

```

x = jit_value_get_param(gcd, 0)
y = jit_value_get_param(gcd, 1)
x_eq_y = jit_insn_eq(gcd, x, y)

' if x = y, return x
Dim label_x_ne_y As jit_label_t = jit_label_undefi
jit_insn_branch_if_not(gcd, x_eq_y, @label_x_ne_y)
jit_insn_return(gcd, x)

' else if...
jit_insn_label(gcd, @label_x_ne_y)

' check x < y
Dim As jit_value_t x_lt_y
Dim label_x_gte_y As jit_label_t = jit_label_undef
x_lt_y = jit_insn_lt(gcd, x, y)
jit_insn_branch_if_not(gcd, x_lt_y, @label_x_gte_y)

' if x < y, return gcd(x, y-x)
Dim As jit_value_t gcd_args(0 To 2), gcd_result
gcd_args(0) = x
gcd_args(1) = jit_insn_sub(gcd, y, x)
gcd_result = jit_insn_call( _
    gcd,          _ ' where we are calling from
    "gcd",        _ ' function name
    gcd,          _ ' function reference
    0,            _ ' signature = auto
    @gcd_args(0), _ ' arguments
    2,            _ ' argument count
    0             _ ' flags = nothing special
)
jit_insn_return(gcd, gcd_result)

' else...
jit_insn_label(gcd, @label_x_gte_y)

' return gcd(x-y, y)
gcd_args(0) = jit_insn_sub(gcd, x, y)
gcd_args(1) = y

```

```

gcd_result = jit_insn_call( _
    gcd,          _ ' where we are calling from
    "gcd",       _ ' function name
    gcd,         _ ' function reference
    0,           _ ' signature = auto
    @gcd_args(0), _ ' arguments
    2,           _ ' argument count
    0            _ ' flags = nothing special
)
jit_insn_return(gcd, gcd_result)

' compile function
jit_function_compile(gcd)
jit_context_build_end(context)

' call function
Dim As jit_uint a=21, b=14, result
Dim As jit_uint Ptr args(0 To 1) = {@a, @b}
jit_function_apply(gcd, @args(0), @result)
Print Using "gcd(&, &) = &"; a; b; result

' clean up libjit
jit_context_destroy(context)

```

Lua



Lightweight, embeddable scripting engine using the Lua language.

Website: <http://www.lua.org/>

Platforms supported: Win32, Linux, DOS

Headers to include: Lua/lua.bi

Header version: 5.2.3

Examples: in examples/other-languages/Lua/

Embeddable javascript engine.

Website: <http://www.mozilla.org/js/spidermonkey/>

Platforms supported: Win32, Linux

Headers to include: spidermonkey/jsapi.bi

Header version: from 2006

Example

```
' ' Evaluating javascript code
#include once "spidermonkey/jsapi.bi"

Dim Shared As JSClass global_class = _
(
    @ "global", 0, _
    @JS_PropertyStub, @JS_PropertyStub, @JS_PropertyStub,
    @JS_EnumerateStub, @JS_ResolveStub, @JS_ConvertStub
)

Dim As JSRuntime Ptr rt = JS_NewRuntime(1048576 / 1024)
Dim As JSContext Ptr cx = JS_NewContext(rt, 4096 / 1024)
Dim As JSObject Ptr global = JS_NewObject(cx, @global_class)

JS_InitStandardClasses(cx, global)

' ' This string could also be read in from a file called test_script.c
Const TEST_SCRIPT = _
    !"function fact(n)                \n" + _
    !"{"                               \n" + _
    !"    if (n <= 1)                  \n" + _
    !"        return 1;                 \n" + _
    !"    \n"                           \n" + _
    !"    return n * fact(n - 1); \n" + _
    !"}"                               \n" + _
    !" \n"                             \n" + _
    !"    fact(5)                       \n"
```

```

Dim As jsval rval
If (JS_EvaluateScript(cx, global, TEST_SCRIPT, Len
    Print "JS_EvaluateScript failed"
    Sleep
    End 1
End If

Print "result: " & *JS_GetStringBytes(JS_ValueToSt

JS_DestroyContext(cx)
JS_DestroyRuntime(rt)

```

```

'' Callback example: Functions that are used by th
'' but are implemented in FB.
#include once "spidermonkey/jsapi.bi"

Dim Shared As JSClass global_class = _
( _
    @"global", 0, _
    @JS_PropertyStub, @JS_PropertyStub, @JS_Proper
    @JS_EnumerateStub, @JS_ResolveStub, @JS_Conver
)

Private Function print_callback cdecl _
( _
    ByVal cx As JSContext Ptr, _
    ByVal obj As JSObject Ptr, _
    ByVal argc As uintN, _
    ByVal argv As jsval Ptr, _
    ByVal rval As jsval Ptr _
) As JSBool

If (argc < 1) Then
    Return 0
End If

```

```

        Print *JS_GetStringBytes(JS_ValueToString(cx,
        Return 1
End Function

Private Function ucase_callback cdecl _
( _
    ByVal cx As JSContext Ptr, _
    ByVal obj As JSObject Ptr, _
    ByVal argc As uintN, _
    ByVal argv As jsval Ptr, _
    ByVal rval As jsval Ptr _
) As JSBool

If (argc < 1) Then
    Return 0
End If

'' Get the first argument
Dim As ZString Ptr arg1 = JS_GetStringBytes(JS

'' Get a buffer for the result string
Dim As ZString Ptr result = JS_malloc(cx, Len(

'' Do the work
*result = UCase(*arg1)

'' Return it in rval
*rval = STRING_TO_JSVAL(JS_NewString(cx, resul

Return 1
End Function

Dim As JSRuntime Ptr rt = JS_NewRuntime(104857
Dim As JSContext Ptr cx = JS_NewContext(rt, 40
Dim As JSObject Ptr global = JS_NewObject(cx,

JS_InitStandardClasses(cx, global)

```

```
JS_DefineFunction(cx, global, "print", @print_
JS_DefineFunction(cx, global, "ucase", @ucase_

Const TEST_SCRIPT = "print(ucase('hello'));"

Dim As jsval rval
If (JS_EvaluateScript(cx, global, TEST_SCRIPT,
    Print "JS_EvaluateScript failed"
    Sleep
    End 1
End If

JS_DestroyContext(cx)
JS_DestroyRuntime(rt)
```

A powerful security toolkit which allows even inexperienced crypto programmers to use cryptographic services to their software.

Website: <http://www.cs.auckland.ac.nz/~pgut001/cryptlib/>

Platforms supported: Win32, Linux

Headers to include: cryptlib.bi

Header version: from 2005

Examples: in examples/math/cryptlib/

Example

```
#include once "cryptlib.bi"

Function calc_hash( ByVal filename As String, ByVal
    Const BUFFER_SIZE = 8192
    Dim As Byte buffer( 0 To BUFFER_SIZE-1 )

    ' create a new context using the wanted algo
    Dim As CRYPT_CONTEXT ctx
    cryptCreateContext( @ctx, CRYPT_UNUSED, algo )

    ' open input file in binary mode
    Dim As Integer f = FreeFile()
    If( Open( filename For Binary Access Read As #
        Return ""
    End If

    ' read until end-of-file
    Do Until( EOF( f ) )
        Dim As Integer oldpos = Seek( f )
        Get #f, , buffer()
        Dim As Integer readlength = Seek( f ) - oldpos
        ' encrypt
        cryptEncrypt( ctx, @buffer(0), readlength
    Loop
```

```

'' close input file
Close #f

'' finalize
cryptEncrypt( ctx, 0, 0 )

'' get the hash result
Dim As Integer buffersize = BUFFER_SIZE
cryptGetAttributeString( ctx, CRYPT_CTXINFO_HA

'' convert to hexadecimal
Dim As String result = ""
For i As Integer = 0 To buffersize-1
    result += Hex( buffer(i) )
Next

'' free the context
cryptDestroyContext( ctx )

Return result
End Function

Dim As String filename = Trim( Command(1) )
If( Len( filename ) = 0 ) Then
    Print "Usage: hash.exe filename"
    End -1
End If

'' init cryptlib
cryptInit( )

'' calculate hashes
Print "md5: "; calc_hash( filename, CRYPT_ALGC
Print "sha: "; calc_hash( filename, CRYPT_ALGC

'' shutdown cryptlib
cryptEnd( )

Sleep

```


UUID library



The UUID library can be used to generate universally unique identifiers. It's part of the e2fsprogs utilities.

Website: <http://e2fsprogs.sourceforge.net/>

Platforms supported: Win32, Linux

Headers to include: uuid.h

Header version: from 2010

Library for using arbitrarily large integers. Note: This library seems to be possible alternative is **gmp**.

Website: http://valyala.narod.ru/big_int/ (Russian) [the site apparently is

Platforms supported: Win32, Linux

Headers to include: big_int/big_int.bi

Header version: from 2005

Examples: in examples/math/big_int/

Example

```
#include once "big_int/big_int_full.bi"

Sub print_num(ByVal num As big_int Ptr)
  Dim As big_int_str Ptr s = big_int_str_create(
  If (s = 0) Then
    Exit Sub
  End If

  If (big_int_to_str(num, 10, s) <> 0) Then
    Exit Sub
  End If

  Print *s->Str;

  big_int_str_destroy(s)
End Sub

Dim As big_int Ptr bignum = big_int_create(1)

big_int_from_int(2, bignum)
big_int_pow(bignum, 65536, bignum)

Print "2^65536 = ";
print_num(bignum)
Print
```

```
big_int_destroy(bignum)
```

Chipmunk Physics



Chipmunk is a physics library designed specifically for 2D games.

Website: <http://chipmunk-physics.net/>

Platforms supported: Win32, Linux

Headers to include: chipmunk/chipmunk.bi

Header version: 4.1.0

gmp, The GNU Multiple Precision Arithmetic Library

Free library for arbitrary precision arithmetic, operating on signed integers, and floating point numbers.

Website: <http://www.gmp.org>

Platforms supported: Win32, Linux

Headers to include: gmp.h

Header version: 4.1.4

Example

```
#include once "gmp.h"

Dim As mpz_ptr bignum = Allocate(SizeOf(__mpz_struct))
mpz_init_set_si(bignum, 2)
mpz_pow_ui(bignum, bignum, 65536)

Print "2^65536 = ";
Dim As ZString Ptr s = mpz_get_str(0, 10, bignum)
Print *s;
Deallocate(s)
Print

mpz_clear(bignum)
Deallocate(bignum)
```

gsl, The GNU Scientific Library



Provides a wide range of mathematical routines such as random number

Website: <http://www.gnu.org/software/gsl/>, Windows port: <http://gnuwin32.org/>

Platforms supported: Win32, Linux

Headers to include: `gsl/*.bi`

Header version: 1.6

Examples: in `examples/math/GSL/`

Example

```
' ' Elementary math example
#include "gsl/gsl_math.bi"

' ' Raise the value of 3.141 to the fourth power
? "3.141 ^ 4 = "; gsl_pow_4(3.141)
?

' ' Find the hypotenuse of a right triangle with sides a and b
? "The hypotenuse of a right triangle with sides a and b is c"
?

Sleep
```

```
' ' Matrix example
#include "gsl/gsl_matrix.bi"

' ' gsl uses the c-style row major order, unlike VBA
? "A 3x3 matrix"
Dim As gsl_matrix Ptr m = gsl_matrix_alloc(3, 3)
For i As Integer = 0 To 2
    For j As Integer = 0 To 2
        gsl_matrix_set (m, i, j, 0.23 + 100*i + j)
    Next
Next
```

```
Next
For i As Integer = 0 To 2
    For j As Integer = 0 To 2
        Print "m(";i;",";j;) = "; gsl_matrix_get
    Next
Next
?
gsl_matrix_transpose(m)
? "And its transpose"
For i As Integer = 0 To 2
    For j As Integer = 0 To 2
        Print "m(";i;",";j;) = "; gsl_matrix_get
    Next
Next
Sleep
```

Newton



The Newton Physics Engine is an integrated solution for real time simulation of physics environments.

Website: <http://newtondynamics.com/>

Platforms supported: Win32, Linux

Headers to include: Newton.bi

Header version: from 2005

Examples: in examples/math/Newton/

ODE, the Open Dynamics Engine



Open source, high performance library for simulating rigid body dynamics.

Website: <http://www.ode.org/>

Platforms supported: Win32, Linux

Headers to include: ode/ode.bi

Header version: 0.11.1

Examples: in examples/math/ODE/

Small C library for creating CGI programs for Websites.

Website: <http://www.newbreedsoftware.com/cgi-util/>

Platforms supported: Win32, Linux

Headers to include: `cgi-util.bi`

Free and easy-to-use client-side URL transfer library supporting almost

Website: <http://curl.haxx.se/libcurl/>
Platforms supported: Win32, Linux, DOS
Headers to include: curl.bi
Header version: 7.39.0
Examples: in examples/network/curl/

Example

```
' ' Curl HTTP Get example

#include once "curl.bi"
#include once "crt/string.bi"

' ' this callback will be called when any data is r
Private Function write_callback cdecl _
( _
    ByVal buffer As Byte Ptr, _
    ByVal size As Integer, _
    ByVal nitems As Integer, _
    ByVal outstream As Any Ptr _
) As Integer

Static As ZString Ptr zstr = 0
Static As Integer maxbytes = 0

Dim As Integer bytes = size * nitems

' ' current zstring buffer too small?
If( maxbytes < bytes ) Then
    zstr = Reallocate( zstr, bytes + 1 )
    maxbytes = bytes
End If

' ' "buffer" is not null-
```

```
terminated, so we must dup it and add the null-ter
    memcpy( zstr, buffer, bytes )
    zstr[bytes] = 0

    ' ' just print it..
    Print *zstr

    Return bytes
End Function

    ' ' init
    Dim As CURL Ptr curl = curl_easy_init( )
    If( curl = 0 ) Then
        End 1
    End If

    ' ' set url and callback
    curl_easy_setopt( curl, CURLOPT_URL, "freebasi
    curl_easy_setopt( curl, CURLOPT_WRITEFUNCTION,

    ' ' execute..
    curl_easy_perform( curl )

    ' ' shutdown
    curl_easy_cleanup( curl )
```

Open extension to CGI that provides high performance without the limitations of specific APIs.

Website: <http://www.fastcgi.com>

Platforms supported: Win32, Linux

Headers to include: fastcgi/fastcgi.bi, fastcgi/fcgiapp.bi, fastcgi/fcgi_stdio.bi

Header version: 2.4.1-SNAP-0311112127

Example

```
#include "fastcgi/fcgi_stdio.bi"

Dim As Integer count = 0
While (FCGI_Accept() >= 0)
    count += 1
    Print !"Content-type: text/html\r\n"
    Print !"\r\n"
    Print "<title>FastCGI Hello!</title>"
    Print "<h1>FastCGI Hello!</h1>"
    Print Using "Request number ### running on host %s\r\n" & *getenv("SERVER_NAME"); count; *getenv("SERVER_NAME");
Wend
```

High-performance asynchronous messaging library

Website: <http://www.zeromq.org/>

Platforms supported: Win32, Linux

Headers to include: `zmq/zmq.h`

Header version: 2.1.10

Stream oriented XML parser library with several useful features.

Website: <http://expat.sourceforge.net/>

Platforms supported: Win32, Linux

Headers to include: expat.bi

Header version: 1.95.8

Examples: in examples/xml/

Example

```
' XML file parser command line tool based on libe
' Can use zstring or wstring (libexpat or libexpa
#define XML_UNICODE

#include once "expat.bi"

#define FALSE 0
#define NULL 0

Const BUFFER_SIZE = 1024

Type Context
  As Integer nesting
  As XML_char * (BUFFER_SIZE+1) text
  As Integer textlength
End Type

Dim Shared As Context ctx

' Callback called by libexpat when begin of XML t
Sub elementBegin cdecl _
  ( _
    ByVal userdata As Any Ptr, _
    ByVal element As XML_char Ptr, _
    ByVal attributes As XML_char Ptr Ptr _
```

```

)

'' Show its name
Print Space(ctx.nesting);*element;

'' and its attributes (attributes are given as
'' much like argv, for each attribute there wi
'' element representing the name and a second
'' specified value)
While (*attributes)
    Print " ";**attributes;
    attributes += 1
    Print "='";**attributes;"";
    attributes += 1
Wend
Print

ctx.nesting += 1
ctx.text[0] = 0
ctx.textlength = 0
End Sub

'' Callback called by libexpat when end of XML tag
Sub elementEnd cdecl(ByVal userdata As Any Ptr, By
'' Show text collected in charData() callback
Print Space(ctx.nesting);ctx.text
ctx.text[0] = 0
ctx.textlength = 0
ctx.nesting -= 1
End Sub

Sub charData cdecl _
( _
    ByVal userdata As Any Ptr, _
    ByVal chars As XML_char Ptr, _ '' Note: r
    ByVal length As Integer _
)

'' This callback will apparently receive every

```

```

'' (really?), including newlines and space.

'' Append to our buffer, if there still is free space
If (length <= (BUFFER_SIZE - ctx.textlength))
    fb_MemCopy(ctx.text[ctx.textlength], chars, length)
    ctx.textlength += length
    ctx.text[ctx.textlength] = 0
End If
End Sub

''
'' Main
''

Dim As String filename = Command(1)
If (Len(filename) = 0) Then
    Print "Usage: expat <xmlfilename>"
    End 1
End If

Dim As XML_Parser parser = XML_ParserCreate(NULL)
If (parser = NULL) Then
    Print "XML_ParserCreate failed"
    End 1
End If

XML_SetUserData(parser, userdata_pointer)
XML_SetElementHandler(parser, @elementBegin, @elementEnd)
XML_SetCharacterDataHandler(parser, @charData)

If (Open(filename, For Input, As #1)) Then
    Print "Could not open file: ";filename;"
    End 1
End If

Static As UByte buffer(0 To (BUFFER_SIZE-1))

Dim As Integer reached_eof = FALSE

```

```

Do
    Dim As Integer size = BUFFER_SIZE
    Dim As Integer result = Get(#1, , buffer(0)
    If (result Or (size <= 0)) Then
        Print "File input error"
        End 1
    End If

    reached_eof = (EOF(1) <> FALSE)

    If (XML_Parse(parser, @buffer(0), size, re
        Print filename & "
(" & XML_GetCurrentLineNumber(parser) & "): Error
        Print *XML_ErrorString(XML_GetErrorCod
        End 1
    End If
Loop While (reached_eof = FALSE)

XML_ParserFree(parser)

```

De-facto standard library for accessing xml files.

Website: <http://xmlsoft.org/>

Platforms supported: Win32, Linux

Headers to include: libxml/*.bi

Header version: 2.6.17

Examples: in examples/xml/

Example

```
#include once "libxml/xmlreader.bi"
#define NULL 0

Dim As String filename = Command(1)
If( Len( filename ) = 0 ) Then
    Print "Usage: libxml filename"
    End 1
End If

Dim As xmlTextReaderPtr reader = xmlReaderForFile(
If (reader = NULL) Then
    Print "Unable to open "; filename
    End 1
End If

Dim As Integer ret = xmlTextReaderRead( reader )
Do While( ret = 1 )
    Dim As ZString Ptr constname = xmlTextReaderCo
    Dim As ZString Ptr value = xmlTextReaderConstV

    Print xmlTextReaderDepth( reader ); _
          xmlTextReaderNodeType( reader ); _
          " "; *constname; _
          xmlTextReaderIsEmptyElement(reader); _
          xmlTextReaderHasValue( reader ); _
          *value
```

```
    ret = xmlTextReaderRead( reader )  
Loop  
  
xmlFreeTextReader( reader )  
  
If( ret <> 0 ) Then  
    Print "failed to parse: "; filename  
End If  
  
xmlCleanupParser( )  
xmlMemoryDump()
```

XSLT itself is a an XML language to define transformation for XML.

Website: <http://xmlsoft.org/XSLT/>

Platforms supported: Win32, Linux

Headers to include: libxslt/libxslt.bi

Header version: 1.1.13

mxml, Mini-XML



Mini-XML is a small XML parsing library that you can use to read XML and XML-like data files in your application without requiring large non-standard libraries.

Website: <http://www.minixml.org/>

Platforms supported: Win32, Linux

Headers to include: mxml.bi

Header version: 2.7

PCRE, Perl Compatible Regular Expressions



Consists of a set of functions that implement regular expression pattern matching using the same syntax and semantics as Perl 5.

Website: <http://www.pcre.org>

Platforms supported: Win32, Linux

Headers to include: pcre.bi, pcre16.bi, pcreposix.bi

Version: 8.31

Examples: in examples/regex/PCRE/

TRE (regex matching library)



Lightweight, robust, and efficient POSIX compliant regexp matching library

Website: <http://laurikari.net/tre/>

Platforms supported: Win32, Linux

Headers to include: `tre/tre.bi`, `tre/regex.bi`

Header version: 0.8.0

Examples: in `examples/regex/TRE/`

libbzip2



libbzip2 is the library implementing .bz2 file or in-memory compression and extraction, with interfaces similar to zlib.

Website: <http://bzip.org/>

Platforms supported: Win32, Linux, DOS

Headers to include: bzlib.bi

Header version: 1.0.6

Examples: in examples/compression/

Easy-to-use library for creating, reading out or modifying .zip archives.

Website: <http://www.nih.at/libzip/>

Platforms supported: Win32, Linux, DOS

Headers to include: zip.bi

Header version: 0.11.2

Examples: in examples/compression/

Example

```
' ' .zip unpacking using libzip
#include once "zip.bi"

Sub create_parent_dirs(ByVal file As ZString Ptr)
    ' Given a path like this:
    ' foo/bar/baz/file.ext
    ' Do these mkdir()'s:
    ' foo
    ' foo/bar
    ' foo/bar/baz
    Dim As UByte Ptr p = file
    Do
        Select Case (*p)
        Case Asc("/")
            *p = 0
            Mkdir(*file)
            *p = Asc("/")
        Case 0
            Exit Do
        End Select
        p += 1
    Loop
End Sub

' ' Asks libzip for information on file number 'i'
' ' and then extracts it, while creating directorie
```

```

Private Sub unpack_zip_file(ByVal zip As zip Ptr,
    #define BUFFER_SIZE (1024 * 512)
    Static As UByte chunk(0 To (BUFFER_SIZE - 1))
    #define buffer (@chunk(0))

    ' Retrieve the filename.
    Dim As String filename = *zip_get_name(zip, i,
    Print "file: " & filename & ", ";

    ' Retrieve the file size via a zip_stat().
    Dim As zip_stat stat
    If (zip_stat_index(zip, i, 0, @stat)) Then
        Print "zip_stat() failed"
        Return
    End If

    If ((stat.valid And ZIP_STAT_SIZE) = 0) Then
        Print "could not retrieve file size from z
        Return
    End If

    Print stat.size & " bytes"

    ' Create directories if needed
    create_parent_dirs(filename)

    ' Write out the file
    Dim As Integer fo = FreeFile()
    If (Open(filename, For Binary, Access Write, A
        Print "could not open output file"
        Return
    End If

    ' Input for the file comes from libzip
    Dim As zip_file Ptr fi = zip_fopen_index(zip,
    Do
        ' Write out the file content as returned
        ' also does the decoding and everything.
        ' zip_fread() fills our buffer

```

```

    Dim As Integer bytes = _
        zip_fread(fi, buffer, BUFFER_SIZE)
    If (bytes < 0) Then
        Print "zip_fread() failed"
        Exit Do
    End If

    '' EOF?
    If (bytes = 0) Then
        Exit Do
    End If

    '' Write <bytes> amount of bytes of the fi
    If (Put(#fo, , *buffer, bytes)) Then
        Print "file output failed"
        Exit Do
    End If
Loop

'' Done
zip_fclose(fi)
Close #fo
End Sub

Sub unpack_zip(ByRef archive As String)
    Dim As zip Ptr zip = zip_open(archive, ZIP_CHECKSUM)
    If (zip = NULL) Then
        Print "could not open input file " & archive
        Return
    End If

    '' For each file in the .zip... (really nice API)
    For i As Integer = 0 To (zip_get_num_entries(zip, 0)) - 1
        unpack_zip_file(zip, i)
    Next

    zip_close(zip)
End Sub

```

```
unpack_zip("test.zip")
```

Configurable compression library based around the LZMA algorithm with zlib-like API. liblzma is the heart of the xz-utils used to handle the .lzma and .xz file formats. It is based on 7-Zip's LZMA SDK.

Website: <http://tukaani.org/xz/>

Platforms supported: Win32, Linux, DOS

Headers to include: lzma.h

Header version: 5.0.2

Examples: in examples/compression/

LZO is a compression library offering fast compression and very fast de

Website: <http://www.oberhumer.com/opensource/lzo/>

Platforms supported: Win32, Linux, DOS

Headers to include: lzo/lzo.bi

Header version: 2.02

Example

```
#include "lzo/lzo1x.bi"

Dim inbuf As ZString Ptr = @"string to compress (c
Dim inlen As Integer = Len(*inbuf) + 1
Dim complen As lzo_uint = 100
Dim compbuf As ZString Ptr = Allocate(complen)
Dim decompflen As lzo_uint = 100
Dim decompbuf As ZString Ptr = Allocate(decompflen)
Dim workmem As Any Ptr

Print "initializing LZ0: ";
If lzo_init() = 0 Then
    Print "ok"
Else
    Print "failed!"
    End 1
End If

Print "compressing '" & *inbuf & "': ";

workmem = Allocate(LZ01X_1_15_MEM_COMPRESS)

If lzo1x_1_15_compress(inbuf, inlen, compbuf, @com
    Print "ok (" & inlen & " bytes in, " & complen
Else
    Print "failed!"
    End 1
```

```
End If

Deallocate(workmem)

Print "decompressing: ";

workmem = Allocate(LZ01X_MEM_DECOMPRESS)

If lzo1x_decompress(compbuf, complen, decompbuf, @
    Print "ok: '" & *decompbuf & "' (" & complen &
Else
    Print "failed!"
    End 1
End If

Deallocate(workmem)
```

Cross-platform fast compression library

Website: <http://www.quicklz.com>

Platforms supported: Win32, Linux, DOS

Headers to include: quicklz.bi

Header version: 1.5.0

Example: examples/compression/QuickLZ.bas

Loss-less data compression library using the Deflate algorithm unencur

Website: <http://www.zlib.net>

Platforms supported: Win32, Linux, DOS

Headers to include: zlib.bi

Header version: 1.2.8

Examples: in examples/compression/

Example

zlib-based PNG save & load code: <http://www.freebasic.net/forum/view>

In-memory compression example:

```
' Zlib compress/decompress example, by yetifoot
#include once "zlib.bi"

Dim As Integer errlev

' This is the size of our test data in bytes.
Dim As Integer src_len = 100000

Print "ZLib test - Version " & *zlibVersion()
Print
Print "Test data size      : " & src_len & " bytes

' The size of the destination buffer for the comp
' the compressBound function.
Dim As Integer dest_len = compressBound(src_len)

' Allocate our needed memory.
Dim As UByte Ptr src = Allocate(src_len)
Dim As UByte Ptr dest = Allocate(dest_len)

' Fill the src buffer with random, yet still comp
For i As Integer = 0 To src_len - 1
```

```

        src[i] = Rnd * 4
    Next

    ' Store the crc32 checksum of the input data, so
    ' uncompression has worked.
    Dim As UInteger crc = crc32(0, src, src_len)

    ' Perform the compression.  dest_len is passed as
    ' the function returns it will contain the size of
    errlev = compress(dest, @dest_len, src, src_len)
    If errlev <> 0 Then
        ' If the function returns a value other than
        Print "**** Error during compress - code " & errlev
    End If
    Print "Compressed to          : " & dest_len & " bytes"

    ' NOTE: in normal use in a program, you would store
    ' be able to tell uncompress the output size.  However,
    ' just leave it in src_len.  The same goes for dest_len
    ' datas size.

    ' Wipe the src buffer before we uncompress to it,
    ' decompression has worked.
    For i As Integer = 0 To src_len - 1
        src[i] = 0
    Next

    ' Perform a decompression.  This time we uncompress
    ' src_len is passed as its address, because when
    ' the function returns it will contain the size of
    errlev = uncompress(src, @src_len, dest, dest_len)
    If errlev <> 0 Then
        ' If the function returns a value other than
        Print "**** Error during uncompress - code " & errlev
    End If
    Print "Uncompressed to        : " & src_len & " bytes"

    ' Make sure the checksum of the uncompressed data
    If crc <> crc32(0, src, src_len) Then

```

```
    Print "crc32 checksum      : FAILED"
Else
    Print "crc32 checksum      : PASSED"
End If

'' Free the buffers used in the test.
Deallocate(src)
Deallocate(dest)

Print
Print "Press any key to end . . ."
Sleep
```

CRT, the C Runtime Library



Standard C language functions. On Windows, this is implemented in msvcrt.dll (however, there also are version-specific msvcrtXX.dlls, the Microsoft Visual C++ runtimes). On Linux, the C runtime is typically implemented by glibc. For DOS, FreeBASIC uses DJGPP, which provides a libc library.

Websites: <http://msdn.microsoft.com/en-us/library/59ey50w6.aspx>,
<http://www.gnu.org/software/libc/>, <http://www.delorie.com/djgpp/>

Platforms supported: Win32, Linux, DOS

Headers to include: crt.bi

Function reference: **C Runtime Functions**

MSDN function reference: <http://msdn.microsoft.com/en-us/library/634ca0c2.aspx>

DOS API



Provides access to low-level BIOS and DOS calls.

Website: <http://freedos.org>

Platforms supported: DOS

Headers to include: `dos/dos.bi`

Examples: in `examples/DOS/`

Disphelper is a COM helper library that can be used in plain C. No MFC

Website: <http://disphelper.sourceforge.net/>

Platforms supported: Win32, Linux (using WINE)

Headers to include: disphelper/disphelper.bi

Header version: from 2005

Example

```
' ' HTTP GET example, using MSXML2

#define UNICODE
#include "disphelper/disphelper.bi"

DISPATCH_OBJ(objHTTP)

dhInitialize(TRUE)
dhToggleExceptions(TRUE)

dhCreateObject("MSXML2.XMLHTTP.4.0", NULL, @objHTT

dhCallMethod(objHTTP, ".Open(%s, %s, %b)", "GET",
dhCallMethod(objHTTP, ".Send")

Dim As ZString Ptr szResponse
dhGetValue("%s", @szResponse, objHTTP, ".ResponseT

Print "Response: "; *szResponse
dhFreeString(szResponse)

SAFE_RELEASE(objHTTP)
dhUninitialize(TRUE)
```

```

'' IExplorer example

#define UNICODE
#include "disphelper/disphelper.bi"

Sub navigate(ByRef url As String)
    DISPATCH_OBJ(ieApp)
    dhInitialize(TRUE)
    dhToggleExceptions(TRUE)

    dhCreateObject("InternetExplorer.Application",
    dhPutValue(ieApp, "Visible = %b", TRUE)
    dhCallMethod(ieApp, ".Navigate(%s)", url)

    SAFE_RELEASE(ieApp)
    dhUninitialize(TRUE)
End Sub

navigate("www.freebasic.net")

```

```

'' VB Script example

#define UNICODE
#include "disphelper/disphelper.bi"

'' This function runs a script using the MSScriptC
'' Optionally returns a result.
Sub RunScript _
    ( _
        ByVal result_identifier As LPWSTR, _
        ByVal result As LPVOID, _
        ByVal script As LPWSTR, _
        ByVal language As LPWSTR _
    )

```

```

DISPATCH_OBJ(control)
If (SUCCEEDED(dhCreateObject("MSScriptControl.
    If (SUCCEEDED(dhPutValue(control, ".Language
        dhPutValue(control, ".AllowUI = %b", T
        dhPutValue(control, ".UseSafeSubset =

        If (result) Then
            dhGetValue(result_identifier, resu
        Else
            dhCallMethod(control, ".Eval(%T)",
        End If
    End If
End If

SAFE_RELEASE(control)
End Sub

dhInitialize(TRUE)
dhToggleExceptions(TRUE)

' VBScript sample
RunScript(NULL, NULL, !"MsgBox(\"This Is a VBS

' JScript sample
Dim As Integer result
RunScript("%d", @result, "Math.round(Math.pow(
Print "Result ="; result

Print "Press any key to exit..."
Sleep

dhUninitialize(TRUE)

```

Universal utility library most commonly used with **GTK+** and GNOME. Provides a main loop implementation for event-based programming, portable multi-threading, portable file/pipe I/O, many utilities such as command line parsing, timers, XML parsing, regular expressions, Unicode manipulation, and also general-purpose data structures.

Website: <http://developer.gnome.org/glib/>

Platforms supported: Linux, Win32

Headers to include: glib.h

Version: 2.42.2

Examples: in examples/misc/glib/

Installing FreeBASIC, any additionally needed packages, and perhaps a text editor or IDE.

Windows 32bit

- Download the latest **FreeBASIC-x.xx.x-win32.exe** installer
- Run it and click through it. The installer will install FreeBASIC at `C:\%ProgramFiles%\FreeBASIC`, or if you chose a different installation directory, in your chosen directory. Start Menu shortcuts to the website will be installed as well.
- Unless you already have a source code editor or IDE, you should install one too, as FreeBASIC itself does not include one. An IDE can be used to write and save .bas files and to launch the FreeBASIC Compiler to compile them. The following IDEs are known to explicitly support FreeBASIC:
 - **FBIDE**
 - **FBEdit**

To uninstall FreeBASIC, remove it from the system's list of installed software (**Add/remove programs, Uninstall or change a program**).

Windows x64

- Download the latest **FreeBASIC-x.xx.x-win64.zip** package
- Extract it where you like, for example at `C:\%ProgramFiles%\FreeBASIC` (no further installation required to use fbc).
- You may want to install a source code editor or IDE; also see the **Windows 32bit** section.

To uninstall FreeBASIC, simply deleted the directory where you extracte it.

Linux

- Download the latest **FreeBASIC-x.xx.x-linux-x86.tar.gz (32bit)** or **FreeBASIC-x.xx.x-linux-x86_64.tar.gz (64bit)** package
- Extract the archive, for example by doing right-click -> Extract Here, or manually in a terminal:

```
$ cd Downloads
$ tar xzf FreeBASIC-x.xx.x-linux-x86.tar.gz
```

- The FreeBASIC compiler can be used from where it was extracted. Usually it is installed into the `/usr/local` system directory though, so that the `fbcc` program is available through-out the whole system. To do that, run the included installation script:

```
$ cd FreeBASIC-x.xx.x-linux-x86
$ sudo ./install.sh -i
```

The `install.sh` script can also be given a path as in `./install.sh -i /usr` if you prefer to install into a directory other than the default `/usr/local`. This default is a good choice though, as it avoids mixing with the content of `/usr` which is usually managed by the distribution's packaging tool.

- FreeBASIC requires several additional packages to be installed before it can be used to compile executables. In general, these are:
 - `binutils`
 - `libc` development files (installing `gcc` will typically install these too)
 - `gcc`
 - `libncurses` development files
 - `X11` development files (for FB graphics programs)
 - `libffi` development files (for the **Threadcall** keyword)
 - `gpm` (general purpose mouse) daemon and `libgpm`

(only needed for **GetMouse** support in the Linux console)

The actual package names to install vary depending on the GNU/Linux distribution.

For native development (32bit FB on 32bit system, or 64bit FB on 64bit system):

- Debian/Ubuntu:
 - gcc
 - libncurses5-dev
 - libffi-dev
 - libgl1-mesa-dev
 - libx11-dev libxext-dev libxrender-dev libxrandr-dev libxpm-dev
- Fedora:
 - gcc
 - ncurses-devel
 - libffi-devel
 - mesa-libGL-devel
 - libX11-devel libXext-devel libXrender-devel libXrandr-devel libXpm-devel
- OpenSUSE:
 - gcc
 - ncurses-devel
 - libffi46-devel
 - xorg-x11-devel

For 32bit development on a 64bit system:

- Debian/Ubuntu:
 - gcc-multilib
 - lib32ncurses5-dev
 - libx11-dev:i386 libxext-dev:i386 libxrender-dev:i386 libxrandr-dev:i386 libxpm-dev:i386
 - (See comment below re Ubuntu 10.04 LTS)

- OpenSUSE:
 - gcc-32bit
 - ncurses-devel-32bit
 - xorg-x11-devel-32bit
 - xorg-x11-libX11-devel-32bit
 - xorg-x11-libXext-devel-32bit
 - xorg-x11-libXrender-devel-32bit
 - xorg-x11-libXpm-devel-32bit
 - libffi46-devel-32bit
- Unless you already have a text editor or IDE, you should install one too, as FreeBASIC itself does not include one. An IDE can be used to write and save .bas files and to launch the FreeBASIC Compiler to compile them. The following IDEs are known to explicitly support FreeBASIC:
 - **Geany**

To uninstall FreeBASIC from `/usr/local`, you can run the `install.sh` script again, but with the `-u` option: `sudo ./install.sh -u`

DOS

- Download the latest **FreeBASIC-x.xx.x-dos.zip** archive
- Find a place for FreeBASIC with at least 13 MiB free space.
- Unpack the ZIP archive, making sure that the directory structure as used inside the archive is preserved ("PKUNZIP -d" for example).
- The top-level directory is named `FreeBASIC-x.xx.x-dos` (will be truncated to "FREEBAS1" in DOS without full LFN support), so you might want to rename it then to a convenient DOS-compliant name not longer than 8 characters and containing no white-spaces, like "FB".
- All the important files used by the compiler (includes, libs) inside the archive do have DOS-compliant names, therefore DOSLFN is not required to use FreeBASIC, however, some examples and texts do have longer names and will be truncated when extracted without full LFN support.

(Note: you can install the DOS version "over" the Windows one or vice-versa, or "merge" those installations later, but rename the FBC.EXE file of the previous installation to FBCW.EXE , FBCD.EXE or such, or it will be overwritten by the new one. Other platform specific files are placed in subdirectories making sure that they won't conflict.)

Compiling under Ubuntu 10.04 LTS, 64-bit:

This comment applies to FB 1.01.0, and may apply to other builds also. Install all of the Libraries listed above; some of the entries ending in ":i386" may throw "not found" errors.

To verify that you're using a 64-bit build, use: "uname -a" or "uname -m" (it'll show x86_64 for 64-bit, i386 for 32-bit).

Then, when running FBC, an error may appear: "error while loading shared libraries: libtinfo.so.5: cannot open shared object file: No such file or directory".

"libtinfo.so.5" is available as a separate library in Ubuntu 11.10+, but it is built into "ncurses.so.5" in 10.04 LTS. So, we need to re-direct the libtinfo references into the ncurses.so.5 libraries:

- Issue: find / -name 'libtinfo.so.5' - just to verify that there are no confusing references to these libraries anywhere. Any references should be checked, and probably deleted?
- Change to the folder containing the FBC executable (perhaps "/usr/local/bin/").
- Issue: ldd fbc - it will list the various library folder(s) being searched (probably "/lib32" in most cases).
- Issue: sudo ln -s /lib32/libncurses.so.5 /lib32/libtinfo.so.5 (assuming "/lib32" was emitted in the previous step).
- Issue: sudo ln -s /lib32/libtinfo.so.5 /lib32/libtinfo.so (assuming "/lib32" ...)
- Retry!
- [Unrelated point: if "private" Libraries are needed for compiles, they were expected to be in /usr/local/lib/freebasic/. Now, they may have to be in /usr/local/lib/freebasic/linux-x86/].
- [Mike Kennedy, Jan, 2015. (This note was not acceptable as a standard "comment" - I don't know why?).

See also

- **Invoking the Compiler**
- **Compiler Command Line Options**

Windows version

- The FreeBASIC compiler (fbc.exe) and the executables generate by it, need at least Windows 98 to run.
- The msvcrt.dll (the Microsoft's C runtime library) must be present (note: it wasn't shipped with Windows 95, but it's installed by many applications and can be also downloaded at: [Microsoft](#)).
- The gfx routines will use DirectX 5.0 or later if found on the host system, otherwise they'll fall back on standard Win32 GDI which will work on any Windows system.
- Unicode wide strings (WSTRING's) only work in Windows NT/2000/XP/2003/Vista or above. Applications that depend on wide-strings will run in Windows 98/Me, but no input/output will work if the character set isn't Latin-based, because those platforms don't support Unicode strings. Windows 95 has most Unicode API functions missing; applications using wide strings won't even be loaded by this specific OS.

Linux version

- The FreeBASIC compiler (fbc) and the executable generated by it depend on libc, libm, libpthread, libdl and libncurses. These are a standard Linux libraries and should be available by default on all modern distros.
- When using the gfx routines, the dependencies will increase. FreeBASIC gfx programs will also need libX11, libXext, libXpm, libXrender and libXrandr to be installed on the host system to be executed. This is usually not a problem as long as there's a recent X11 server installed in the system (at least XFree86 4.3.0 or any X.org version).
- If having a working X11 installation is enough to run FreeBASIC gfx programs, it may be not enough to compile them; you may need to install the X11 development libraries from your Linux packages repository.
- Unicode wide-strings (WSTRING's) with non-ASCII character set can only be displayed in console if the locale is set to an UTF-8

version - most modern distros come with support that and char sets other than latin may work only in xterm.

DOS version

- Official requirement: A DPMI (DOS Protected-Mode Interface) server must be present to run fbc.exe and any executable generated by it. This is not as bad as it looks. It simply means, the "CWSDPMI.EXE" file (cca 20 KiB) must be present in the same directory or a place where the PATH environment variable points to. CWSDPMI package: homer.rice.edu...cwsdpmi (note: FreeDOS comes with it already installed). Further, there is a possibility to bypass this problem, and to use alternatively HDPMI for details see [DOS related FAQ](#) .
- You need a 80386 or newer CPU and cca 4 MiB of RAM. For compiling of large programs or libraries, you will need more. Similar applies to executables generated by FBC, those using FB's graphics library however will need a better/faster CPU (200 MHz (?), work in progress, code not yet fully optimized, and exact minimum not known by now). FBC and executables generated by it need an FPU (80387, 80487, always built-in since Pentium). This requirement can be bypassed using "EMU387" (auto-loaded if needed, but not included in FB packages, see delorie.com/djgpp/...), or by avoiding floats and (non-trivial) removing float-related startup code.
- The DOS version should run in any DOS, like FreeDOS, [Enhanced-]DR-DOS (do **not** use the DR-EMM386's included DPMI, use CWSDPMI or HDPMI), or MS-DOS. It also works properly under a number of "DOS box" environments that emulate a DOS system, such as the Windows NTVDM; however, some of these environments are not implemented faithfully and contain bugs, so caution should be exercised.
- Long filenames are supported under systems that supply the long filename API defined by Windows 95, including DOS with an LFN TSR (for example DOSLFN **(1)** **(2)**). Long filename support is not required to use the compiler; however, care must be taken in unpacking the distribution, for example, with a Windows program which creates short names with numeric tails (FREEBA~1) instead of truncating to 8 characters (FREEBAS1). The filenames of all

files in the distribution should be truncated to 8.3 if the compiler is to be used without long filename support.

- There are a few limitations, see [DOS related FAQ](#) .

See also

- [Installing FreeBASIC](#)
- [Compiler Command Line Options](#)

and

- [Compiler FAQ](#)
- [Win32 related FAQ](#)
- [DOS related FAQ](#)
- [LINUX related FAQ](#)

Invoking the compiler after installation.

Windows

The compiler can be manually invoked from the command-line, or automatically by your IDE/Code Editor. If you're using an IDE, you will usually have to tell it where the compiler was installed, so it can find it. How exactly to do that depends on the IDE.

To compile manually, you should append the FreeBASIC installation directory to your PATH environment variable, separating it from previous entries using a semi-colon. Now you can simply use "fbc" from the command prompt, instead of always having to type in the full path (e.g. "C:\FreeBASIC\fbc.exe").

Then, open a console/command prompt/MS DOS prompt, in the same directory as your program. To compile your program, you can use:

```
C:\mystuff\myprogram\> fbc myprogram.bas
```

and `myprogram.exe` will be created in the same directory.

A console can be launched in a specific directory from Explorer by using Microsoft's "Open Command Window Here" PowerToy on Windows XP. On Windows Vista & above you can SHIFT+RightClick on a folder in Explorer to see the 'Open Command Window Here' option. As a last resort, you can also select Start -> Run, type "cmd" and hit Enter, and use the "cd" command to change the current directory.

Note: You can in fact invoke the compiler from any directory you like, but you have to specify the correct path to your program, so the compiler can find it, for example:

```
C:\> fbc mystuff\myprogram\myprogram.bas
```

The resulting executable will still be put in the same directory as the

program.

Linux

If the `install.sh` script was successfully executed with enough privileges the compiler binary should have been copied `/usr/local/bin/fbc`, allowing any user access to the compiler from any directory.

From the prompt, type,

```
fbc
```

to see a list of options. To compile the "Hello, world!" example program, navigate to the directory where the FreeBASIC examples were installed (`/usr/local/share/freebasic`), and type,

```
fbc examples/misc/hello.bas
```

and a `./hello` executable file will be created in the `examples/misc` directory.

Linux (standalone)

If the `install` script `install-standalone.sh` was successfully executed with enough privileges, a link to the compiler binary should have been created at `/usr/bin/fbc`, allowing any user access to the compiler from any directory. If it was not possible to create the link, you may want to alter your `PATH` environmental variable to be able to invoke the compiler from any directory. Navigate to the directory where FreeBASIC was installed.

From the prompt, type,

```
fbc
```

to see a list of options. To compile the "Hello, world!" example program type,

```
fbc examples/misc/hello.bas
```

and a `./hello` executable file will be created in the `examples/misc` directory.

DOS

Navigate to the directory where FreeBASIC was installed. For example, FreeBASIC is installed in the directory `C:\FB`, type,

```
C:  
CD FB
```

Some DOSes accept "`CDD C:\FB`" as well. You can also add the FreeBASIC directory to your PATH environment variable (usually something like "`SET PATH=C:\FB\;%PATH%`") so you can invoke the compile from any directory.

At the prompt, type,

```
fb
```

to see a list of options. To compile the "Hello, world!" example program type,

```
fb examples\misc\hello.bas
```

and a `hello.exe` executable file will be created in the `examples\misc` directory.

See also

- [Installing FreeBASIC](#)
- [Compiler Command Line Options](#)
- [Compiler FAQ](#)

Using the **fbc** command-line.

The official FreeBASIC distribution comes with **fbc**, FreeBASIC's flagship compiler. **fbc** is a command line compiler, and can be launched from the console - from DOS, the Windows command prompt or a Linux shell. Running **fbc** from the console without any arguments displays a list of available options, or command-line switches, that can be used to adjust the behavior of the compiler.

At its simplest, **fbc** takes a source file as a command-line argument and produces an executable file. It does this by compiling the source file (.bas) into an assembly (.asm) file, then compiling this into an object file (.o) using GAS and finally linking using LD this object file to other object files and libraries it needs to run, producing the final executable file. The assembly and compiled object files are deleted at this point by default. For example, the following command,

```
fbc foo.bas
```

produces the executable `foo.exe` in DOS and Windows, and `./foo` in Linux. **fbc** can accept multiple source files at once, compile and link them all into one executable. For example, the following command,

```
fbc foo.bas bar.bas baz.bas
```

produces the executable `foo.exe` in DOS and Windows, and `./foo` in Linux. Since `foo.bas` was listed first, it will be the main entry point into the executable, and also provide its name. To specify a different entry point or executable name, use the "-m" and "-x" switches, respectively. To have, for example, `baz.bas` provide the main entry point into an executable called `foobar.exe`, you would use

```
fbc -x foobar.exe -m baz foo.bas bar.bas baz.bas
```

The "-x" switch names the executable verbatim, so in Linux, the executable produced from the above command would be called `./foobar.exe`.

Syntax

`fbc [options] [input_list]`

Where *input_list* is a list of filenames. Accepted files are:

File extension	Description
.bas	FreeBASIC source file
.a	Library
.o	Object file
.rc	Resource script (Windows only)
.res	Compiled resource (Windows only)
.xpm	X icon pixmap (Linux only)

Source code

-b < name >

Add a source file to compilation

-i < name >

Add a path to search for include files

-include < name >

Include a header file on each source compiled

-d < name=val >

Add a preprocessor's define

-lang < name >

Select language mode: fb, fb1ite, qb, deprecated

-forcelang < name >

Select language mode: fb, fb1ite, qb, deprecated (overrides statements in code)

Code generation

-target < platform >

Linking

-a < name >

Add an object file to linker's list

-l < name >

Add a library file to linker's list

-p < name >

Add a path to search for libraries

-mt

Link with thread-safe runtime library

-nodeflibs

Do not include the default libraries

-static

Prefer static libraries over dynamic ones when linking

-map < name >

Save the linking map to file name

-Wl < opt >

Pass options to LD (separated

Set the target platform for cross compilation

-gen < backend >

Sets the compiler backend (default is 'gas').

-asm < format >

Sets the assembler format for Asm block.

-arch < type >

Set target architecture (default: 486)

-O < level >

Set the optimization level (-gen gcc).

-vec < level >

Set level of vector optimizations enabled by the compiler (default: 0)

-fpu < type >

Set the floating point arithmetics unit (default: FPU)

-fpmode < type >

Select between fast and accurate floating-point operations (default: PRECISE)

-z < value >

Sets miscellaneous or experimental options.

Compilation

-m < name >

Main file without extension, the entry point (default is the first .bas file on the command line)

-g

Add debug info

-profile

Enable function profiling

-e

by commas)

-export

Export symbols for dynamic linkage

-lib

Create a static library

-dylib

Create a DLL, including the import library

-dll

Create a DLL, including the import library. (Same as -dylib)

-x < name >

Set executable/library path/name

Behaviour

-prefix < path >

Set the compiler prefix path

-version

Show compiler version on the command line, do not compile or link.

-v

Be verbose

-print < option >

Display certain information (host, target, etc.)

-pp

Emit the preprocessed input file only, do not compile

-r

Compile into intermediate file(s) only, do not assemble or link

-rr

Compile into asm file(s) only, do not assemble or link

-c

Compile and assemble source

Add error checking

-ex

Add error checking with
RESUME support

-exx

Same as **-ex** plus array bounds
and null-pointer checking

-Wa < opt >

Pass options to GAS (separated
by commas)

-Wc < opt >

Pass options to GCC (separated
by commas)

-o < name >

Set object file path/name (must
be passed after the .bas file)

file only, do not link

-R

Do not delete the intermediate
file(s)

-RR

Do not delete the asm file(s)

-C

Do not delete the object file(s)

-w < value >

Set min warning level: all,
pedantic, next or a value

-maxerr < val >

Only stop parsing if <val> errors
occurred

-noerrline

Do not show source line where
error occurred

Target specific

-s < name >

Set subsystem (gui, console)

-t < value >

Set stack size in kbytes (default:
1M)

Meta

@< file >

Read (additional) command-line
options from a file

Example

```
fbc myfile.bas
```

*(With DOS version of FBC, compile and link a DOS executable
MYFILE.EXE.)*

```
fbc -s gui myfile.bas
```

(With Windows version of FBC, compile and link a Windows executable myfile.exe. Running the program will not show the console window ("MS-DOS Prompt"))

```
fbc -lib module1.bas module2.bas module3.bas -x libmylib.a
```

(Compile and link a static library libmylib.a from the three source file:

```
fbc -m main_module -c main_module.bas
```

(Compile an object file main_module.o and mark it as an entry point)

```
fbc -c sub_module.bas
```

(Compile an object file sub_module.o)

```
fbc -x application.exe main_module.o sub_module.o
```

(Link an executable application.exe)

Note: How to include an icon in a FB executable program

There is a simple command line option to compile a FB program into an executable with an icon:

- Create a *.rc file, for example appicon.rc, with this info:

```
FB_PROGRAM_ICON ICON "appicon.ico"
```

(where appicon.ico is the name of icon)

- Then when compiling program, add appicon.rc in the list of files to compile.

See also

- [Compiler Options](#)
- [Installing FreeBASIC](#)
- [Invoking the FreeBASIC compiler](#)

Compiler Options



Command line compiler options for the fbc compiler:

@< file >

- Read (additional) command-line options from the file

-a < name >

- Add an object file to linker's list

-arch < type >

- Set target architecture (default: 486)

-asm < format >

- Sets the assembler format for Asm block

-b < name >

- Add a source file to compilation

-c

- Compile only, do not link

-C

- Do not delete the object file(s)

-d < name=val >

- Add a preprocessor's define

-dll

- Create a DLL, including the import library. (Same as -dylib)

-dylib

- Create a DLL, including the import library

-e

- Add error checking

-ex

- Add error checking with RESUME support

-exx

- Same as -ex plus array bounds and null-pointer checking

-export

- Export symbols for dynamic linkage

-forcelang <name>

- Select language compatibility, overriding #lang/\$lang in

code

-fpmode < type >

- Select between fast and accurate floating-point operations (default: PRECISE)

-fpu < type >

- Set the floating point arithmetics unit (default: FPU)

-g

- Add debug info

-gen < backend >

- Sets the compiler backend (default is 'gas')

-i < name >

- Add a path to search for include files

-include < name >

- Include a header file on each source compiled

-l < name >

- Add a library file to linker's list

-lang < name >

- Select language compatibility: fb, fb1ite, qb, deprecated

-lib

- Create a static library

-m < name >

- Main file without extension, the entry point (default is the first .bas file on the command line)

-map < name >

- Save the linking map to file name

-maxerr < val >

- Only stop parsing if <val> errors occurred

-mt

- Link with thread-safe runtime library

-nodeflibs

- Do not include the default libraries

-noerrline

- Do not show source line where error occurred

-o < name >

- Set object file path/name (must be passed after the .bas

- file)
- O < level >**
 - Set the optimization level (-gen gcc)
- p < name >**
 - Add a path to search for libraries
- pic**
 - Generate position-indepdent code (non-x86 Unix shared libs)
- pp**
 - Emit the preprocessed input file only, do not compile
- prefix < path >**
 - Set the compiler prefix path
- print < option >**
 - Let the compiler display certain information (host, target,
- profile**
 - Enable function profiling
- r**
 - Compile into *.asm/*.c/*.11 file(s) only, do not assemble or link
- R**
 - Preserve intermediate *.asm/*.c/*.11 file(s) generated by compilation
- rr**
 - Compile into *.asm file(s) only, do not assemble or link
- RR**
 - Preserve intermediate *.asm files generated by compilation
- s < name >**
 - Set subsystem (gui, console)
- showincludes**
 - Display a tree of file names of #included files
- static**
 - Prefer static libraries over dynamic ones when linking
- target < platform >**
 - Set the target platform for cross compilation
- t < value >**

- Set stack size in kbytes (default: 1M)
- v**
 - Be verbose
- vec < level >**
 - Set level of vector optimizations enabled by the compiler (default: 0)
- version**
 - Show compiler version
- w < value >**
 - Set min warning level: all, pedantic or a value
- Wa < opt >**
 - Pass options to GAS (separated by commas)
- Wc < opt >**
 - Pass options to GCC (separated by commas)
- Wl < opt >**
 - Pass options to LD (separated by commas)
- x < name >**
 - Set executable/library path/name
- z < value >**
 - Sets miscellaneous or experimental options

See also

- **Using the Command Line**

Compiler Option: @file



Read (additional) command-line options from the file

Syntax

@file

Parameters

file

Name of a text file containing command line options. It's possible to use multiple lines in the file. The options may be separated by spaces or line breaks, and support double-quoted strings to allow spaces in parameters (like the real command line). This file can itself contain additional *@file* options.

Description

The *@file* compiler option tells the compiler to parse the specified file to find more command line options. The options found in the file are treated as if they were found on the command line. This can be useful to pass very long command lines to the compiler, for example on DOS where command lines are limited in length.

Example

options.txt:

```
-d TEST=123
```

opts.bas:

```
Print "TEST=" & TEST
```

Compile with:

```
fbcc @options.txt opts.bas
```

Output:

```
TEST=123
```

See also

- **Using the Command Line**

Compiler Option: -a



Add an object file to the linker's list

Syntax

```
[ -a ] < object file >
```

Parameters

object file

Name of the object file with extension.

Description

The -a compiler option adds a compiled object file to the linker's list. The "-a" is optional if the object file name has a ".o" file extension.

See also

- [Compiler Option: -b](#)
- [Using the Command Line](#)

Compiler Option: -arch



Set target architecture for improved/restricted code generation or cross-compiling

Syntax

`-arch < architecture >`

Parameters

architecture

The target architecture. Recognized values:

- Related to 32bit x86:
 - 386
 - 486 (default for x86)
 - 586
 - 686
 - athlon
 - athlon-xp
 - athlon-fx
 - k8-sse3
 - pentium-mmx
 - pentium2
 - pentium3
 - pentium4
 - pentium4-sse3
- Related to 64bit x86_64:
 - x86_64, x86-64, amd64
- Related to 32bit ARM:
 - armv6
 - armv7-a (default for ARM)
- Related to 64bit ARM (AArch64):
 - aarch64

- Others:
 - `native`: For compiling to the architecture which the compiler is running on.
 - `32, 64`: For quick cross-compiling to the 32bit or 64bit version of the default architecture.

Description

The `-arch` compiler option sets the target CPU architecture. This can be used for multiple purposes:

- Improving code generation; for example: You can use `-arch 686` to override the default `-arch 486`, and the compiler will generate faster code in some cases, by using certain instructions which were not available on i486 (or other CPUs older than i686).
- Restricting code generation; for example: You can use `-arch 386` to limit the compiler to using only i386-compatible instructions.
- Cross-compiling; for example: You can use `-arch x86_64` on 32bit x86 systems to cross-compile to 64bit x86_64.

The exact impact which the `-arch` setting has on code generation depends on the **code generation backend** that is being used. The x86 ASM backend (`-gen gas`) handles the `-arch` setting and adjusts code generation accordingly in some cases. When using the GCC backend (`-gen gcc`), the specified architecture will be passed on to `gcc` via `gcc -march=<...>`, causing `gcc` to generate code for the specified architecture.

However, `-arch` only affects newly generated code, but not pre-compiled code such as the FreeBASIC runtime libraries, or any other library from the `lib/` directory. For example, using `-arch 386` is not necessarily enough to get a pure i386 executable -- it also depends on how all the libraries that will be linked in were compiled.

The `-arch 32` and `-arch 64` shortcuts are similar to `gcc's -m32/-m64`

options. On 32bit architectures, `-arch 64` is an abbreviation for cross-compiling to the default 64bit version of the architecture (e.g. from 32bit x86 to 64bit x86_64, or 32bit ARM to 64bit AArch64), and `-arch 32` does nothing. On 64bit systems, it is the other way round: `-arch 32` cross-compile to the default 32bit architecture, while `-arch 64` does nothing.

The `-arch native` shortcut is similar to gcc's `-march=native` option. On x86, it causes fbc to try and detect the host CPU automatically based on the `cputid` instruction and its availability or results. On other architectures, this will currently simply use the architecture which the compiler itself was built for. Under `-gen gcc` this will use `gcc -march=native`.

Specifying an `-arch` setting incompatible to the native architecture will trigger **cross-compilation**, just like the `-target` option, except that only the target architecture, but not the target operating system, is changed.

See also

- **Using the Command Line**
- **-target**
- **FB and cross-compiling**

Compiler Option: -asm



Set assembler format for inline assembly under -gen gcc

Syntax

`-asm < format >`

Parameters

format

The assembler format: *intel* or *att*

Description

The `-asm` compiler option sets the assembler format for inline `Asm` blocks when using `-gen gcc`.

- `-gen gcc -asm intel`: FB inline assembly blocks must use FB's usual Intel syntax format. Under `-gen gcc`, fbc will try to translate it to gcc's format automatically. For example:

```
Dim a As Long = 1
Print a
Asm
    inc dword Ptr [a]
End Asm
Print a
```

- `-gen gcc -asm att`: FB inline assembly blocks must use **gcc's format**. For example:

```
Dim a As Long = 1
Print a
Asm
    "incl %0\n" : "+m" (a) : :
End Asm
```

Print a

The x86 ASM backend (**-gen gas**) currently only supports `-asm intel` and using `-asm att` results in an error.

See also

- [__Fb_Asm__](#)
- [Using the Command Line](#)

Compiler Option: -b



Add a source file to compilation

Syntax

```
[ -b ] < source file >
```

Parameters

source file

The name with extension, and optionally a path, of the basic source file.

Description

The **-b** option adds a source file to the compilation list. In general, this option is redundant since source files with a *.bas* extension can be specified without it, but is useful if a source file does not have a *.bas* extension, or if exists in an other directory.

To compile and link the source files *file1.bas*, *file2.bas* and *file3.fb* into an executable (*file1.exe* in DOS/Windows, *file1* in Linux), type,

```
fbc -b file1.bas file2.bas -b file3.fb
```

Note that the **-b** option was not needed for *file1.bas* or *file2.bas*.

See also

- [Compiler Option: -a](#)
- [Using the Command Line](#)

Compiler Option: -c



Compile and assemble source file only, do not link

Syntax

-c

Description

The -c option specifies that any source files listed are to be compiled and assembled into object files, and not linked into an executable (the default behavior). When using the "-c" switch, "-m" must be specified when compiling a main source file.

See also

- [Compiler Option: -C](#)
- [Compiler Option: -r](#)
- [Compiler Option: -m](#)
- [Compiler Option: -o](#)
- [Using the Command Line](#)

Compiler Option: -C



Do not delete the object file(s)

Syntax

-c

Description

The -c compiler option causes the object file(s) that are generated during the compile process to not be deleted.

See also

- [Compiler Option: -c](#)
- [Compiler Option: -R](#)
- [Using the Command Line](#)

Compiler Option: -d



Add a preprocessor definition

Syntax

```
-d < name=value >  
-d < name >
```

Parameters

name

Name of the preprocessor macro to define. No parameters are allowed.

value

Value to give to the macro. If omitted, it will be defined as 1

Description

The `-d` compiler option adds a preprocessor macro to all source files. The same as using the preprocessor directive `#define` or `#macro`.

See also

- `#define`
- `#macro`
- **Intrinsic Defines**
- **Using the Command Line**

Compiler Option: -dll



Create a DLL and import library

Syntax

`-dll`

Description

The `-dll` compiler option creates a dynamic link library. This creates a DLL under Windows (including the import library), and creates a `.so` under Linux.

The intrinsic macro `__FB_OUT_DLL__` is set to non-zero (-1) if the `-dll` option was specified, and set to zero (0) otherwise.

Platform Differences

- Not supported on the DOS platform.

See also

- `__FB_OUT_DLL__`
- [Shared Libraries](#)
- [Using the Command Line](#)

Compiler Option: -dylib



Create a DLL and import library

Syntax

`-dylib`

Description

The `-dylib` compiler option creates a dynamic link library. This creates a DLL under Windows (including the import library), and creates a `.so` under Linux.

The intrinsic macro `__FB_OUT_DLL__` is set to non-zero (-1) if the `-d11` option was specified, and set to zero (0) otherwise.

Platform Differences

- Not supported on the DOS platform.

See also

- `__FB_OUT_DLL__`
- [Shared Libraries](#)
- [Using the Command Line](#)

Compiler Option: -e



Add error checking

Syntax

-e

Description

Adds QuickBASIC-like error checking.

See also

- `__FB_ERR__`
- **Compiler Option: -ex**
- **Compiler Option: -exx**
- **Error Handling**
- **Using the Command Line**

Compiler Option: -ex



Add error checking with **Resume** support

Syntax

-ex

Description

The -ex compiler option adds error handling as with the -e option plus support for **Resume**.

See also

- [__FB_ERR__](#)
- [Compiler Option: -e](#)
- [Compiler Option: -exx](#)
- [Error Handling](#)
- [Using the Command Line](#)

Compiler Option: -exx



Add error checking with **Resume** support and array bounds and null-pointer checking

Syntax

-exx

Description

The -exx compiler option adds error checking with **Resume** support plus array bounds and null-pointer checking (including the procedure pointers).

See also

- [__FB_ERR__](#)
- [Compiler Option: -e](#)
- [Compiler Option: -ex](#)
- [Error Handling](#)
- [Using the Command Line](#)

Compiler Option: -export



Export symbols for dynamic linkage

Syntax

`-export`

Description

The `-export` compiler option exports symbols for dynamic linkage.

See also

- [Export](#)
- [Shared Libraries](#)
- [Using the Command Line](#)

Compiler Option: -forcelang



Provides QuickBASIC or backward compatibility

Syntax

`-forcelang dialect`

Parameters

dialect

The dialect to use in compilation, one of `fb` (default), `fb1ite`, `qb` or `deprecated`.

Description

The `-forcelang` compiler option changes the way source code is interpreted, and is meant as a tool to users wanting traditional QuickBASIC-like behavior, or behavior deprecated from previous versions of FreeBASIC. It overrides any `#lang` statements within the code.

The intrinsic macro `__FB_LANG__` is set to the string name of the dialect specified on the command line, or `"fb"` by default.

To learn more about the differences between each of these language dialects, see [Compiler Dialects](#).

fb

This is the default dialect, and allows compilation of source code adhering to the most recent version of the FreeBASIC language.

fb1ite

This dialect provides support for FreeBASIC syntax and functionality, but with a more traditional QuickBASIC programming style.

qb

This dialect provides the best support for older QuickBASIC code.

deprecated

This dialect is for backward compatibility with some previous versions of FreeBASIC, however, this dialect may not exist in future versions. Programmers should consider using the "fbLite" dialect instead.

See also

- [#lang](#)
- [__FB_LANG__](#)
- [Compiler Option: -lang](#)
- [Compiler Dialects](#)
- [Using the Command Line](#)

Compiler Option: -fpmode



Selects faster, less accurate or slower, more precise floating-point math.

Syntax

`-fpmode < mode >`

Parameters

mode

The floating point mode: FAST | PRECISE.

Description

The `-fpmode` compiler option specifies whether speed or precision is more important for floating point math. If this option is not specified, the default is `-fpmode PRECISE`.

`-fpmode FAST` will generate faster, less accurate instructions for certain floating point operations.

`-fpmode PRECISE` will generate standard floating point instructions that operate at the default speed and accuracy of the selected floating point unit.

Currently, the only floating point operations that behave differently when using `-fpmode FAST` are: **Sin()**, **Cos()**, reciprocal, and reciprocal **Square Root**, all of which must operate on **single** precision values.

Using `-fpmode PRECISE` is dependent on the `-fpu SSE` command line option. Using `-fpmode PRECISE` without using `-fpu SSE` will generate an error.

See also

- **Using the Command Line**
- **Compiler Option: -fpu**
- `__Fb_Fpmode__`

Compiler Option: -fpu



Sets the math unit to be used for floating point arithmetics.

Syntax

`-fpu < type >`

Parameters

type

The floating point unit: x87 | SSE.

Description

The `-fpu` compiler option sets the math unit to be used for floating point arithmetics. If this option is not specified, the default is `-fpu x87`.

`-fpu x87` will generate floating point instructions for the 387.

`-fpu SSE` will generate floating point instructions for SSE and SSE2 with some math support still done by the 387.

Functions normally return a floating point value (**Single** or **Double**) in the `st(0)` register. Sometimes, this may be optimized by returning the value in the `xmm0` register instead. This can be specified with `option("Sse")` after the return type in a function's declaration or definition. `option("Sse")` is ignored unless the source is compiled with the `-fpu SSE` command line option.

See also

- [Using the Command Line](#)
- [Option\(\)](#)
- [__Fb_Fpu__](#)

Compiler Option: -g



Add debug information

Syntax

-g

Description

The `-g` compiler option inserts debugging symbols into output files, to use with GDB-compatible debuggers.

The intrinsic macro `__FB_DEBUG__` is set to non-zero (-1) if the option was specified, and set to zero (0) otherwise.

See also

- `__FB_DEBUG__`
- [Debugging](#)
- [Using the Command Line](#)

Compiler Option: -gen



Sets the backend code emitter.

Syntax

-gen < *backend* >

Parameters

backend

gas for x86 GAS assembly, **gcc** for GNU C, **llvm** for LLVM IR.

Description

The **-gen** compiler option sets the backend code emitter and assembler.

-gen gas

The compiler will emit GAS assembler code to a `.asm` file which will then be compiled to an object file using 'as'. This is fbc's original x86 code generation backend.

-gen gcc

The compiler will emit C code to a `.c` file which will then be compiled to an `.asm` file using 'gcc' as a high level assembler. The C backend is intended to make FB portable to more platforms than just x86. This requires gcc to be installed so that fbc can invoke it to compile the C code, also see [Installing gcc for -gen gcc](#).

-gen llvm

The compiler will emit LLVM IR code to a `.ll` file which will then be compiled to an `.asm` file using 'llc'. The LLVM backend is still a work in progress. It is intended for the same purpose as the C backend, and could theoretically solve some of the C backend's problems, such as debugging meta data support.

See also

- [Tools used by fbc](#)
- [Using the Command Line](#)

Compiler Option: -i



Add a path to search for include files

Syntax

```
-i < include path >
```

Parameters

include path

The directory path, relative or absolute, of where to search for include files.

Description

The `-i` option allows an additional directory to be used when searching for header files. By default, `fbcc` searches in the current directory, and *prefix/inc* in that order--where, *prefix* is the location where FreeBASIC is installed. A directory specified with the `-i` option will be searched before these default directories, and when the `-i` option is used multiple times, `fbcc` will search the directories in the order they are listed on the command-line.

To search in the subdirectory *includes* first for header files while compiling the source file *file.bas*, type,

```
fbcc -i includes file.bas
```

See also

- `#include`
- [Compiler Option: -include](#)
- [Header Files](#)
- [Using the Command Line](#)

Compiler Option: -include



Include a header file on each source compiled

Syntax

```
-include < include file >
```

Parameters

include file

The header file name with extension, and optionally a path, to include

Description

The `-include` option has the effect of adding an `#include` preprocessor directive to the very beginning of each source file before processing it. When used multiple times, the files will be included in the order they are listed on the command-line.

To include the file *header.bi* when processing *file1.bas* and *file2.bas*, type,

```
fbcc -include header.bi file1.bas file2.bas
```

See also

- `#include`
- [Compiler Option: -i](#)
- [Header Files](#)
- [Using the Command Line](#)

Compiler Option: -l



Add a library file to the linker's list.

Syntax

`-l < libname >`

Parameters

libname

Name of the library to link in. The library file name's extension should not be included. For example, when using `-l something`, the linker will look for the files:

- `Libsomething.a`
- `Libsomething.dll.a` (Windows)
- `something.dll` (Windows)
- `Libsomething.so` (Linux)

Description

The `-l` compiler option adds a library file to the linker's list, to be linked into the final executable or library if needed to satisfy dependencies.

See also

- [#includ](#)
- [Compiler Option: -p](#)
- [Using the Command Line](#)

Compiler Option: -lang



Provides QuickBASIC or backward compatibility

Syntax

`-lang dialect`

Parameters

dialect

The dialect to use in compilation, one of `fb` (default), `fb1ite`, `qb` or `deprecated`.

Description

The `-lang` compiler option changes the way source code is interpreted and is meant as a tool to users wanting traditional QuickBASIC-like behavior, or behavior deprecated from previous versions of FreeBASIC.

The intrinsic macro `__FB_LANG__` is set to the string name of the dialect specified on the command line, or "fb" by default.

To learn more about the differences between each of these language dialects, see [Compiler Dialects](#).

fb

This is the default dialect, and allows compilation of source code adhering to the most recent version of the FreeBASIC language.

fb1ite

This dialect provides support for FreeBASIC syntax and functionality, but with a more traditional QuickBASIC programming style.

qb

This dialect provides the best support for older QuickBASIC code.

deprecated

This dialect is for backward compatibility with some previous versions of FreeBASIC, however, this dialect may not exist in future versions. Programmers should consider using the "fbLite" dialect instead.

Note: this command-line option can be overridden by any **#lang** statements used in the code.

See also

- **#lang**
- **__FB_LANG__**
- **Compiler Option: -forcelang**
- **Compiler Dialects**
- **Using the Command Line**

Compiler Option: `-lib`



Create a static library

Syntax

`-lib`

Description

The `-lib` compiler option creates a static library.

The intrinsic macro `__FB_OUT_LIB__` is set to non-zero (-1) if the `-lib` option was specified, and set to zero (0) otherwise.

See also

- `__FB_OUT_LIB__`
- [Static Libraries](#)
- [Using the Command Line](#)

Compiler Option: -m



Main file without extension to indicate the main module

Syntax

`-m < source file >`

Parameters

source file

The name without extension of the main module source file

Description

The `-m` compiler option specifies a main entry point for a source file; the argument is the name of a source file minus its extension. If `"-m"` is not specified, the first source file listed is given a main entry point. When using the `"-c"` or `"-r"` switch, `"-m"` must be specified when compiling a main source file.

The intrinsic macro `__FB_MAIN__` is defined in the main module and not defined in other modules.

See also

- `__FB_MAIN__`
- [Compiler Option: -c](#)
- [Compiler Option: -r](#)
- [Using the Command Line](#)

Compiler Option: -map



Save the linking map to file name

Syntax

```
-map < map file >
```

Parameters

map file

Name of the map file to save generated during linking.

Description

The `-map` compiler option saves the a map file of the executable made

See also

- [Using the Command Line](#)

Compiler Option: -maxerr



Set maximum number of errors to report before aborting compilation

Syntax

```
-maxerr < value | "inf" >
```

Parameters

value | "inf"

Specifies the maximum number of errors or no maximum if "inf" is given instead of a value.

Description

The `-maxerr` compiler option sets the maximum number of errors the compiler must find before stopping. The default is 10. If **inf**, for infinite is specified the compiler continues until it finds the end of the source. Useful if an IDE is parsing the error messages.

See also

- [Using the Command Line](#)

Compiler Option: -mt



Link with thread-safe runtime library

Syntax

`-mt`

Description

The `-mt` compiler option forces linking with thread-safe runtime library for multithreaded applications. The thread-safe version is always used automatically if the FreeBASIC built-in threading functions are used, so you only need to specify this option if using your own threading routines.

The intrinsic macro `__FB_MT__` is set to non-zero (-1) if the `-mt` option was specified, and set to zero (0) otherwise.

See also

- `__FB_MT__`
- [Using the Command Line](#)

Compiler Option: `-nodeflibs`



Do not include the default libraries

Syntax

`-nodeflibs`

Description

The `-nodeflibs` compiler option causes default libraries not to be used when linking. The libraries which are normally linked by default can still be used, but only if they are explicitly specified.

See also

- [Using the Command Line](#)

Compiler Option: -noerrline



Do not show source line where error occurred

Syntax

`-noerrline`

Description

The `-noerrline` compiler option causes reported errors to not show the place in source where error occurred. Useful if an IDE is parsing the error messages.

See also

- [Using the Command Line](#)

Compiler Option: -o



Set object file path/name

Syntax

`-o < output file >`

Parameters

output file

The name, with optional path, of the object file to create.

Description

The `-o` option can be used to specify the file name for the object file created while compiling an input file. By default, the name for the object file (and other temporaries like assembly files) is based on the name of the corresponding input file, but with an `.o` extension. This option is useful for example in combination with `-c`, or to force the compiler to create temporary object files in other directories (if, for example, the source code directory is or should be treated as read-only).

Given `-o` options are only assigned to input files that need to be compiled, namely `*.bas`, `*.rc`, `*.res` and `*.xpm`.

Note: `-o` options can appear in front of or behind the input file they correspond to, but there cannot be multiple `-o` options for one input file. For example, these are all accepted:

```
fbc 1.bas -o 1.o
fbc -o 1.o 1.bas
fbc 1.bas -o 1.o 2.bas -o 2.o
fbc 1.bas -o 1.o -o 2.o 2.bas
```

However, this is an error:

```
fbc 1.bas 2.bas -o 1.o -o 2.o
```

The `-v` option makes the compiler show the actual file names that it uses.

See also

- **Compiler Option: -b**
- **Compiler Option: -c**
- **Using the Command Line**

Compiler Option: -O



Set the optimization level for GCC

Syntax

`-O < level >`

Parameters

level

The optimization level: 0, 1, 2, 3 Or max (3).

Description

Specifies the optimization level to be passed to GCC when using `-gen gcc`.

See also

- [Compiler Option: -gen gcc](#)
- [Using the Command Line](#)

Compiler Option: -p



Add a path to search for libraries

Syntax

`-p < library path >`

Parameters

library path

The directory path, relative or absolute, of where to search for library files.

Description

The `-p` compiler option adds a path to search for libraries. By default, libraries are looked for in the system FreeBASIC libraries directory and in the current directory.

See also

- `#libpath`
- [Using the Command Line](#)

Compiler Option: `-pic`



Generate position-independent code (non-x86 Unix shared libs)

Syntax

`-pic`

Description

The `-pic` compiler option tells the compiler to generate position-independent code. This is needed for creating shared libraries on x86_64 or ARM Linux/BSD platforms except Win64 (and also not on 32bit x86). This option should not be used when creating executables (as opposed to shared libraries) though.

By default, `-pic` is enabled when using `-dll` or `-dylib`, and disabled for all other compilation modes. Usually you only have to specify `-pic` you are using `-c` or `-lib` and want to link them into shared libraries later.

`-pic` is implemented by passing `-fPIC` to gcc (when using the `-gen gcc` backend). The `-gen gas` backend does not support position-independent code since it only supports 32bit x86 and there is no special position-independent code needed for shared libraries on 32bit x86.

See also

- [Using the Command Line](#)

Compiler Option: -pp



Emit the preprocessed input file only, do not compile

Syntax

-pp

Description

The `-pp` compiler option enables the preprocessor-only mode. The code is parsed & checked as usual, but is not compiled. A pre-processed version of every input `source.bas` is generated, named `source.pp.bas`.

See also

- [Using the Command Line](#)

Compiler Option: -prefix



Set the compiler prefix path

Syntax

`-prefix < path >`

Parameters

path

The directory, relative or absolute to where fbc is located.

Description

The `-prefix` compiler option sets the compiler prefix (where the compiler finds the bin, lib, and inc directories); and defaults to the path where fbc resides, if this can be determined.

See also

- [Using the Command Line](#)

Compiler Option: -print



Print out information

Syntax

`-print option`

Description

The `-print` option can be used to query the compiler for certain information which may be useful especially for build scripts. It does not prevent compilation of input files given besides the `-print` option, but the compiler also can be invoked with only a `-print` option and no input files, in which case it will not compile anything but only respond to the `-print` option.

Currently, the following `-print` options are recognized:

option	effect
host	Prints the host system on which fbc is running
target	Prints the target system for which fbc is compiling (can be affected by the <code>-target</code> option)
x	Prints the file name of the output executable or library that fbc will or would generate (named after the <code>-x</code> option), depending on other command line options

Example

A **makefile** could use `target := $(shell $(FBC) -print target)` to find out the compilation target, which would even work when cross-compiling, with `FBC` set to something like `fbc -target foo`.

`fbc -print x` alone will print out the executable file extension for the target system.

`fbc -print x -dll` on the other hand will print out the dynamic library file name format.

`fbc -print x -m foo` will print out the executable file name that would

be used when compiling a module called foo.bas.

`fbc 1.bas 2.bas -lib -print x` will compile 1.bas and 2.bas into a library, whose file name will be displayed.

See also

- [-X](#)
- [-target](#)
- [Using the Command Line](#)

Compiler Option: `-profile`



Enable function profiling

Syntax

`-profile`

Description

The `-profile` compiler option enables function profiling. After running an executable compiled with this option, a `gmon.out` file will be created in the program directory, allowing use of GPROF for analysis of the program's execution.

See also

- [Profiling](#)
- [Using the Command Line](#)

Compiler Option: -r



Compile into *.asm/*.c/*.11 file(s) only, do not assemble or link

Syntax

-r

Description

The -r option specifies that any source files listed are to be compiled to *.asm/*.c/*.11 files, depending on the used **code generation backend**, and not compiled or linked into an executable.

When using the -r option, -m must be specified when compiling the main module.

Use the -R option to preserve intermediate files without affecting compilation/assembling/linking.

Use the -rr option to compile input source files to *.asm regardless of the code generation backend.

See also

- [Compiler Option: -c](#)
- [Compiler Option: -R](#)
- [Compiler Option: -m](#)
- [Compiler Option: -gen](#)
- [Compiler Option: -rr](#)
- [Using the Command Line](#)

Compiler Option: -R



Preserve intermediate *.asm/*.c/*.ll file(s) generated by compilation

Syntax

-R

Description

The **-R** compiler option causes the intermediate *.asm/*.c/*.ll file(s) that are generated during the compile process to be preserved. Other than that, compilation is performed as usual. Which files are generated exactly depends on the used **code generation backend** and compilation target.

When compiling a Windows DLL, **-R** also preserves the intermediate *.def file used for generating the import library for the DLL.

See also

- [Compiler Option: -C](#)
- [Compiler Option: -r](#)
- [Compiler Option: -gen](#)
- [Using the Command Line](#)

Compiler Option: -rr



Compile into *.asm file(s) only, do not assemble or link

Syntax

-rr

Description

The **-rr** option specifies that any source files listed are to be compiled to *.asm files, and not compiled or linked into an executable. Unlike with the **-r** option, this works regardless of the used **code generation backend**.

When using the **-rr** option, **-m** must be specified when compiling a main source file.

Use the **-RR** option to preserve the generated *.asm files without affecting compilation/assembling/linking.

See also

- **Compiler Option: -c**
- **Compiler Option: -r**
- **Compiler Option: -RR**
- **Compiler Option: -m**
- **Compiler Option: -gen**
- **Using the Command Line**

Compiler Option: -RR



Preserve intermediate *.asm files generated by compilation

Syntax

-RR

Description

The **-RR** compiler option causes the intermediate *.asm file(s) that are generated during the compile process to be preserved. Other than that, compilation is performed as usual.

See also

- [Compiler Option: -C](#)
- [Compiler Option: -rr](#)
- [Compiler Option: -R](#)
- [Using the Command Line](#)

Compiler Option: -s



Sets the executable subsystem

Syntax

`-s < subsystem >`

Parameters

subsystem

The executable subsystem: `gui` or `console`.

Description

The `-s` compiler option specifies the executable subsystem. Allowed subsystems are `gui` and `console` (by default, `console` is used). Specifying a `gui` subsystem prevents the console window from appearing behind the program window.

Platform Differences

- Supported on Windows and Cygwin only.

See also

- [Using the Command Line](#)

Compiler Option: -showincludes



Display a tree of file names of #included files

Syntax

-showincludes

Description

Tells the compiler to display the file names of loaded source code files in form of a tree. This includes the *.bas files at the toplevel, aswell as the names of #included files as they are being #included. This is intended to be used for debugging #include dependencies, etc.

See also

- [Using the Command Line](#)

Compiler Option: `-static`



Prefer static libraries over dynamic ones when linking

Syntax

`-static`

Description

When creating an executable or a shared library/DLL, the `-static` compiler option can be used to tell the compiler to prefer linking against static libraries rather than shared libraries/DLLs. That way, if the linker finds both static and shared versions of a library, it will use the static version, rather than defaulting to the shared version.

Installing the proper static libraries and then using `-static` can be used to avoid some or all dependencies on shared libraries.

Platform Differences

- On Linux & co it is possible to create purely statically linked executables, because static versions of the system libraries used by FreeBASIC are available.
- On Windows, there are no static versions of the system libraries, but `-static` can still be useful to decide between static library or DLL versions of other libraries, if both are installed.

See also

- [Using the Command Line](#)

Compiler Option: -target



Set the target platform for cross compilation

Syntax

`-target < platform >`

Parameters

platform

The target platform. Recognized values:

- dos
- win32
- win64
- xbox
- <os>-<arch>

<os> can be one of:

- linux
- cygwin
- darwin
- freebsd
- netbsd
- openbsd

<arch> can be one of:

- x86
- x86_64
- arm
- aarch64

Examples:

- linux-x86
- linux-x86_64
- linux-arm
- linux-aarch64

- freebsd-x86
 - freebsd-x86_64
 - ...
- For backwards compatibility, the following values are recognized. They will select the corresponding operating system, together with the compiler's default architecture (same as the host), because these values do not specify an architecture explicitly.
 - linux
 - cygwin
 - darwin
 - freebsd
 - netbsd
 - openbsd
- The **Normal** fbc (e.g. FB-linux release) additionally recognizes GNU triplets, for example
 - i686-w64-mingw32
 - x86_64-w64-mingw32
 - i686-pc-linux-gnu
 - arm-linux-gnueabi
 - ...

Description

The `-target` compiler option can be used to create an executable for a platform which is different from the host on which the source code is being compiled and linked. Appropriate libraries and cross compilation tools (assembler, linker) must be installed for cross compilation to work (also see **FB and cross-compiling**).

If `-target <platform>` is given, the compiler will compile programs more or less as if they were compiled on the given platform. This affects which `__FB_*__` operating-system-specific symbol will be pre-defined, the default calling convention, the object and executable file

format (e.g. ELF/COFF), the available runtime libraries and functions, etc.

With a standalone FB setup such as the FB-dos or FB-win32 releases

- Specifying `-target <platform>` causes the compiler to use the compiler tools in the `bin/<platform>/` directory, and target-specific libraries in the `lib/<platform>/` directory. For example, `-target win32` causes the compiler to compile for Win32 and use tools from `bin/win32/` and libraries from `lib/win32/`.
- It is unnecessary (but safe) to specify a `-target` option that matches the host (for example `-target win32` on `win32`). It does not make a difference to the compilation process.
- If `-target` is not specified, the compiler defaults to compiling for the native system. It will then use the compiler tools and libraries from the `bin/` and `lib/` directories corresponding to the native system.

With a normal FB setup such as the FB-linux release:

- Specifying `-target <platform>` causes the compiler to prefix the `<platform>-` string to the executable names of `binutils` and `gcc`. For example, specifying `-target i686-w64-mingw32` causes the compiler to invoke `i686-w64-mingw32-ld` instead of `ld` (same for other tools besides the linker). This allows `fbcc` to integrate with `binutils/gcc` cross-compiler toolchains and matches how cross-compiling tools are typically installed on Linux distributions.
- Note that specifying something like `-target win32` does not usually make sense here. It causes the compiler to try to use `win32-ld` which usually does not exist, because `binutils/gcc` toolchains for cross-compilation to Windows typically have names such as `i686-pc-mingw32`, not just `win32`. Thus, it is necessary to specify something like `-target i686-pc-mingw32` instead of `-target win32`.

- For backwards compatibility, if the given *platform* string describes the host and is an FB target name (the values accepted by the `-target` option with a standalone FB setup) instead of a GNU triplet, then the `-target` option will be ignored, and the `<platform>-` string will not be prefixed to compiler tools. For example, this allows `-target linux` to work with the FB-linux release. It will be ignored instead of causing the compiler to try to use `linux-ld` instead of `ld`.
- If `-target` is not specified, the compiler defaults to compiling for the native system, and it will invoke `binutils/gcc` without a target-specific prefix. This allows `fbcc` to integrate with usual Linux (and similar) systems where `binutils/gcc` for native compilation are installed without any target-specific prefix.
- Libraries besides FB's own runtime libraries are located by running `gcc -print-file-name=...` (Or `<platform>-gcc -print-file-name=...`). This allows `fbcc` to use the system and `gcc` libraries installed on Linux and similar systems without knowing the exact installation directories.

See also

- **Using the Command Line**
- **FB and cross-compiling**

Compiler Option: -t



Set stack size in kilobytes

Syntax

`-t < stack size >`

Parameters

stack size

Stack size in kilobytes.

Description

The `-t` compiler option sets the stack size in kilobytes (defaults to 102 KBytes). The local arrays are created in the stack, so 1MB of stack is not always enough.

Platform Differences

- Supported on Windows, Cygwin and DOS only.

See also

- [Using the Command Line](#)

Compiler Option: -v



Be verbose

Syntax

-v

Description

The -v compiler option activates verbose mode. In this mode the compiler shows its actions step by step

See also

- [Using the Command Line](#)

Compiler Option: **-vec**



Enables vector optimizations by the compiler.

Syntax

`-vec < level >`

Parameters

level

The level of vectorization: (0 | NONE) | 1 | 2.

Description

The `-vec` compiler option enables multiple levels of optimizations by searching for multiple scalar expressions that can be merged into a single vector expression. If this option is not specified, the default is `-vec 0`.

`-vec 0` | `none` will disable vector optimizations.

`-vec 1` will enable complete expression merging vectorization.

`-vec 2` includes `-vec 1` but also enables intra-expression vectorization

This option is dependent on the `-fpu SSE` command line option. Attempting to enable vector optimizations without using `-fpu SSE` will generate an error.

See also

- [Using the Command Line](#)
- [Compiler option -fpu](#)

Compiler Option: `-version`



Show compiler version

Syntax

`-version`

Description

The `-version` compiler option makes FBC show the compiler version and exit. Any other command-line options are ignored, and no compilation will be performed.

See also

- [Using the Command Line](#)

Compiler Option: -w



Set minimum warning level.

Syntax

`-w level | all | param | Escape | pedantic | Next`

Parameters

level

Warning messages only with a level equal or greater to this value will be output.

all

Equivalent to specifying a *level* of zero (0).

param

Warn when procedure parameters aren't specified with either **ByVal** or **ByRef**.

Escape

Warn when string literals contain any number of escape characters (\).

pedantic

Equivalent to specifying the **param** and **Escape** arguments.

Next

Warn when **Next** is followed by an identifier.

Description

The `-w` compiler option determines which compiler warnings, if any, are output. Each possible warning is associated with a warning level, starting from zero (0) and increasing with the potential problems that may occur. A significantly high *level* value will have the effect of suppressing all warning messages.

Note that the **param**, **Escape**, **pedantic** and **Next** arguments provide additional warnings not ordinarily output, even by default.

If the `-w` option is not specified, it's as if `-w 0` was used. The `-w` option can be specified multiple times.

See also

- **Using the Command Line**

Compiler Option: -Wa



Pass options to the assembler when using the assembly emitter (-gen gas), the default.

Syntax

`-Wa < options >`

Parameters

options

Additional options to pass to the assembler.

Description

The `-wa` compiler option passes additional options to GAS, the assembler. Options must be separated by commas only.

For example:

```
fbc -wa -o,output.o,--verbose
```

See also

- [Compiler Option: -gen](#)
- [Compiler Option: -Wc](#)
- [Compiler Option: -WI](#)
- [Using the Command Line](#)

Compiler Option: -Wc



Pass options to the C compiler when using the C emitter (-gen gcc).

Syntax

`-Wc < options >`

Parameters

options

Additional options to pass to the C compiler.

Description

The `-Wc` compiler option passes additional options to GCC, the C compiler. Options must be separated by commas only.

For example:

```
fbcc -gen gcc -Wc -m32,--verbose,-include,some-header.h
```

See also

- [Compiler Option: -gen](#)
- [Compiler Option: -Wa](#)
- [Compiler Option: -Wl](#)
- [Using the Command Line](#)

Compiler Option: -Wl



Pass options to linker

Syntax

`-Wl < options >`

Parameters

options

Additional options to pass to the linker.

Description

The `-Wl` compiler option passes additional options to LD, the linker. Options must be separated by commas only.

See also

- [Compiler Option: -Wa](#)
- [Using the Command Line](#)

Compiler Option: -x



Set executable/library path/name

Syntax

`-x < name >`

Parameters

name

Name of the executable or library file.

Description

The `-x` compiler option set the executable or library name, with extension. Defaults to the name of the first source file passed on the command line. When compiling libraries, be sure to add the "lib" prefix to the file name, otherwise the linker will not be able to find it. If compiling and linking separately, this option must be set only in the linker.

See also

- [Using the Command Line](#)

Compiler Option: -z



Sets miscellaneous or experimental compiler options.

Syntax

`-z < value >`

Parameters

value

Miscellaneous compiler option.

Description

The `-z` compiler option sets miscellaneous, obscure, temporary, or experimental options used by the developers. There is no guarantee that these options will be supported in future versions of the compiler.

-z gosub-setjmp

Specifies that the `setjmp/longjmp` implementation of `GoSub` should be used even when the GAS backend is used. By default, `GoSub` will be supported in *-gen gas* using `CALL/RET` assembly instructions and in *-gen gcc* using `setjmp/longjmp` C runtime functions.

See also

- [Using the Command Line](#)

The debugger is in the `bin\win32` or `bin\dos` directories (the **GDB.EXE** file), for the Windows and DOS versions respectively. It usually comes already installed in most Linux distros.

(Note: all commands should be typed without quotes and then [return] must be pressed.)

- Compile the sources using the `-g` cmd-line option to add debugging support.
- Load it in GDB using: `"gdb myapplicationname.exe"`
- Set the arguments to the application been debugged using: `"set args arg1 arg2 argn"`. You can also run GDB and pass the arguments directly to the application been debugged: `"gdb --args myapp.exe arg1 arg2 arg3"`.
- If the executable isn't in the same directory of the source files where it was compiled, type: `"dir path/to/my/application/sources"`.
- Place a breakpoint in the first line using: `"b main"`. To place a breakpoint in a function called "abc" use: `"b ABC"` (note: all in uppercase, GDB is case sensitive by default, but you can use the `"set language pascal"` command to change GDB to case-insensitive mode).
- Type `"r"` to start the application.
- Type `"n"` to step over function calls. Keep pressing [return] to skip to the next line.
- Type `"s"` to step into function calls. Same as above.
- Type `"c"` to continue execution until the next breakpoint.
- Use `"print ABC"` to show the contents of the variable called "abc". GDB supports pointer/pointer field dereferencing, indexing and arithmetics too, so `"print *MYPOINTER"` will also work. (note: undeclared variables or the ones with suffixes like `% & ! # $` can't be printed).
- Use `"disp ABC"` to display the contents of a variable called "abc".

- Use "watch ABC" to stop each time a variable called "abc" is changed.
- Use "r" again to restart the application when finished.
- Type "q" to quit.
- Type "help" to see a list of commands, there are many others.

During the program compilation three types of errors can arise:

Compiler Warnings:

The warnings don't stop the compilation, just alert the user some non-recommended and error-prone operation is attempted in the code. Sometimes one of these operations is coded deliberately to achieve a result, in this case the warnings can be disabled by setting the **-w 1** option at the command line.

- *1 Passing scalar as pointer*
- *2 Passing pointer to scalar*
- *3 Passing different pointer types*
- *4 Suspicious pointer assignment*
- *5 Implicit conversion*
- *6 Cannot export symbol without -export option*
- *7 Identifier's name too big, truncated*
- *8 Literal number too big, truncated*
- *9 Literal string too big, truncated*
- *10 UDT with pointer or var-len string fields*
- *11 Implicit variable allocation*
- *12 Missing closing quote in literal string*
- *13 Function result was not explicitly set*
- *14 Branch crossing local variable definition*
- *15 No explicit BYREF or BYVAL*
- *16 Possible escape sequence found in*
- *17 The type length is too large, consider passing BYREF*
- *18 The length of the parameters list is too large, consider passing UDT's BYREF*
- *19 The ANY initializer has no effect on UDT's with default constructors*
- *20 Object files or libraries with mixed multithreading (-mt) options*

- *21 Object files or libraries with mixed language (-lang) options*
- *22 Deleting ANY pointers is undefined*
- *23 Array too large for stack, consider making it var-len or SHARED*
- *24 Variable too large for stack, consider making it SHARED*
- *25 Overflow in constant conversion*
- *26 Variable following NEXT is meaningless*
- *27 Cast to non-pointer*
- *28 Return method mismatch*
- *29 Passing Pointer*
- *30 Command line option overrides directive*
- *31 Directive ignored after first pass*
- *32 'IF' statement found directly after multi-line 'ELSE'*
- *33 Shift value greater than or equal to number of bits in data type*
- *34 '=' parsed as equality operator in function argument, not assignment to BYREF function result*
- *35 Mixing signed/unsigned operands*
- *36 Mismatching parameter initializer*
- *37*
- *38 Mixing operand data types may have undefined results*
- *39 Redefinition of intrinsic*

Compiler Error messages:

The error messages stop the compilation after 10 errors (see the -maxer command-line option to change that default value) or a fatal error occurred, and require a correction by the user before the compilation can be continued. The compiler signals the lines where the errors have been found, so the correction can be done quickly. In a few cases the place pointed at by the error messages is not where the errors can be found, it's the place where the compiler has given up in waiting for something that should be somewhere.

- *1 Argument count mismatch*
- *2 Expected End-of-File*

- 3 *Expected End-of-Line*
- 4 *Duplicated definition*
- 5 *Expected 'AS'*
- 6 *Expected '('*
- 7 *Expected ')'*
- 8 *Undefined symbol*
- 9 *Expected expression*
- 10 *Expected '='*
- 11 *Expected constant*
- 12 *Expected 'TO'*
- 13 *Expected 'NEXT'*
- 14 *Expected identifier*
- 15 *Expected '-'*
- 16 *Expected ','*
- 17 *Syntax error*
- 18 *Element not defined*
- 19 *Expected 'END TYPE' or 'END UNION'*
- 20 *Type mismatch*
- 21 *Internal!*
- 22 *Parameter type mismatch*
- 23 *File not found*
- 24 *Invalid data types*
- 25 *Invalid character*
- 26 *File access error*
- 27 *Recursion level too deep*
- 28 *Expected pointer*
- 29 *Expected 'LOOP'*
- 30 *Expected 'WEND'*
- 31 *Expected 'THEN'*
- 32 *Expected 'END IF'*
- 33 *Illegal 'END'*

- 34 Expected 'CASE'
- 35 Expected 'END SELECT'
- 36 Wrong number of dimensions
- 37 Array boundaries do not match the original EXTERN declaration
- 38 'SUB' or 'FUNCTION' without 'END SUB' or 'END FUNCTION'
- 39 Expected 'END SUB' or 'END FUNCTION'
- 40 Illegal parameter specification
- 41 Variable not declared
- 42 Variable required
- 43 Illegal outside a compound statement
- 44 Expected 'END ASM'
- 45 Function not declared
- 46 Expected ';'
- 47 Undefined label
- 48 Too many array dimensions
- 49 Array too big
- 50 User Defined Type too big
- 51 Expected scalar counter
- 52 Illegal outside a CONSTRUCTOR, DESTRUCTOR, FUNCTION, OPERATOR, PROPERTY or SUB block
- 53 Expected var-len array
- 54 Fixed-len strings cannot be returned from functions
- 55 Array already dimensioned
- 56 Illegal without the -ex option
- 57 Type mismatch
- 58 Illegal specification
- 59 Expected 'END WITH'
- 60 Illegal inside functions
- 61 Statement in between SELECT and first CASE
- 62 Expected array
- 63 Expected '{'

- 64 Expected '}'
- 65 Expected ']'
- 66 Too many expressions
- 67 Expected explicit result type
- 68 Range too large
- 69 Forward references not allowed
- 70 Incomplete type
- 71 Array not dimensioned
- 72 Array access, index expected
- 73 Expected 'END ENUM'
- 74 Var-len arrays cannot be initialized
- 75 '...' ellipsis upper bound given for dynamic array (this is not supported)
- 76 '...' ellipsis upper bound given for array field (this is not supported)
- 77 Invalid bitfield
- 78 Too many parameters
- 79 Macro text too long
- 80 Invalid command-line option
- 81 Selected non-x86 CPU when compiling for DOS
- 82 Selected -gen gas ASM backend for non-x86 CPU
- 83 -asm att used for -gen gas, but -gen gas only supports -asm intel
- 84 -pic used when making executable (only works when making a shared library)
- 85 -pic used, but not supported by target system (only works for non-x86 Unixes)
- 86 Var-len strings cannot be initialized
- 87 Recursive TYPE or UNION not allowed
- 88 Recursive DEFINE not allowed
- 89 Identifier cannot include periods
- 90 Executable not found

- *91 Array out-of-bounds*
- *92 Missing command-line option for*
- *93 Expected 'ANY'*
- *94 Expected 'END SCOPE'*
- *95 Illegal inside a compound statement or scoped block*
- *96 UDT function results cannot be passed by reference*
- *97 Ambiguous call to overloaded function*
- *98 No matching overloaded function*
- *99 Division by zero*
- *100 Cannot pop stack, underflow*
- *101 UDT's containing var-len string fields cannot be initialized*
- *102 Branching to scope block containing local variables*
- *103 Branching to other functions or to module-level*
- *104 Branch crossing local array, var-len string or object definition*
- *105 LOOP without DO*
- *106 NEXT without FOR*
- *107 WEND without WHILE*
- *108 END WITH without WITH*
- *109 END IF without IF*
- *110 END SELECT without SELECT*
- *111 END SUB or FUNCTION without SUB or FUNCTION*
- *112 END SCOPE without SCOPE*
- *113 END NAMESPACE without NAMESPACE*
- *114 END EXTERN without EXTERN*
- *115 ELSEIF without IF*
- *116 ELSE without IF*
- *117 CASE without SELECT*
- *118 Cannot modify a constant*
- *119 Expected period ('.')*
- *120 Expected 'END NAMESPACE'*
- *121 Illegal inside a NAMESPACE block*

- *122 Symbols defined inside namespaces cannot be removed*
- *123 Expected 'END EXTERN'*
- *124 Expected 'END SUB'*
- *125 Expected 'END FUNCTION'*
- *126 Expected 'END CONSTRUCTOR'*
- *127 Expected 'END DESTRUCTOR'*
- *128 Expected 'END OPERATOR'*
- *129 Expected 'END PROPERTY'*
- *130 Declaration outside the original namespace*
- *131 No end of multi-line comment, expected "'/'*
- *132 Too many errors, exiting*
- *133 Expected 'ENDMACRO'*
- *134 EXTERN or COMMON variables cannot be initialized*
- *135 EXTERN or COMMON dynamic arrays cannot have initial bounds*
- *136 At least one parameter must be a user-defined type*
- *137 Parameter or result must be a user-defined type*
- *138 Both parameters can't be of the same type*
- *139 Parameter and result can't be of the same type*
- *140 Invalid result type for this operator*
- *141 Invalid parameter type, it must be the same as the parent TYPE/CLASS*
- *142 Vararg parameters are not allowed in overloaded functions*
- *143 Illegal outside an OPERATOR block*
- *144 Parameter cannot be optional*
- *145 Only valid in -lang*
- *146 Default types or suffixes are only valid in -lang*
- *147 Suffixes are only valid in -lang*
- *148 Implicit variables are only valid in -lang*
- *149 Auto variables are only valid in -lang*
- *150 Invalid array index*
- *151 Operator must be a member function*

- *152 Operator cannot be a member function*
- *153 Method declared in anonymous UDT*
- *154 Constant declared in anonymous UDT*
- *155 Static variable declared in anonymous UDT*
- *156 Expected operator*
- *157 Declaration outside the original namespace or class*
- *158 A destructor should not have any parameters*
- *159 Expected class or UDT identifier*
- *160 Var-len strings cannot be part of UNION's or nested TYPE's*
- *161 Dynamic arrays cannot be part of UNION's or nested TYPE's*
- *162 Fields with constructors cannot be part of UNION's or nested TYPE's*
- *163 Fields with destructors cannot be part of UNION's or nested TYPE's*
- *164 Illegal outside a CONSTRUCTOR block*
- *165 Illegal outside a DESTRUCTOR block*
- *166 UDT's with methods must have unique names*
- *167 Parent is not a class or UDT*
- *168 CONSTRUCTOR() chain call not at top of constructor*
- *169 BASE() initializer not at top of constructor*
- *170 REDIM on UDT with non-CDECL constructor*
- *171 REDIM on UDT with non-CDECL destructor*
- *172 REDIM on UDT with non-parameterless default constructor*
- *173 ERASE on UDT with non-CDECL constructor*
- *174 ERASE on UDT with non-CDECL destructor*
- *175 ERASE on UDT with non-parameterless default constructor*
- *176 This symbol cannot be undefined*
- *177 RETURN mixed with 'FUNCTION =' or EXIT FUNCTION (using both styles together is unsupported when returning objects with constructors)*
- *178 'FUNCTION =' or EXIT FUNCTION mixed with RETURN (using both styles together is unsupported when returning objects with constructors)*

- *179 Missing RETURN to copy-construct function result*
- *180 Invalid assignment/conversion*
- *181 Invalid array subscript*
- *182 TYPE or CLASS has no default constructor*
- *183 Function result TYPE has no default constructor*
- *184 Missing BASE() initializer (base UDT without default constructor requires manual initialization)*
- *185 Missing default constructor implementation (base UDT with default constructor requires manual initialization)*
- *186 Missing UDT.constructor(byref as UDT) implementation (base UDT without default constructor requires manual initialization)*
- *187 Missing UDT.constructor(byref as const UDT) implementation (base UDT without default constructor requires manual initialization)*
- *188 Invalid priority attribute*
- *189 PROPERTY GET should have no parameter, or just one if indexed*
- *190 PROPERTY SET should have one parameter, or just two if indexed*
- *191 Expected 'PROPERTY'*
- *192 Illegal outside a PROPERTY block*
- *193 PROPERTY has no GET method/accessor*
- *194 PROPERTY has no SET method/accessor*
- *195 PROPERTY has no indexed GET method/accessor*
- *196 PROPERTY has no indexed SET method/accessor*
- *197 Missing overloaded operator:*
- *198 The NEW[] operator does not allow explicit calls to constructors*
- *199 The NEW[] operator only supports the { ANY } initialization*
- *200 The NEW operator cannot be used with fixed-length strings*
- *201 Illegal member access*
- *202 Expected ':'*
- *203 The default constructor has no public access*

- *204 Constructor has no public access*
- *205 Destructor has no public access*
- *206 Accessing base UDT's private default constructor*
- *207 Accessing base UDT's private destructor*
- *208 Illegal non-static member access*
- *209 Constructor declared ABSTRACT*
- *210 Constructor declared VIRTUAL*
- *211 Destructor declared ABSTRACT*
- *212 Member cannot be static*
- *213 Member isn't static*
- *214 Only static members can be accessed from static functions*
- *215 The PRIVATE and PUBLIC attributes are not allowed with REDIM, COMMON or EXTERN*
- *216 STATIC used here, but not the in the DECLARE statement*
- *217 CONST used here, but not the in the DECLARE statement*
- *218 VIRTUAL used here, but not the in the DECLARE statement*
- *219 ABSTRACT used here, but not the in the DECLARE statement*
- *220 Method declared VIRTUAL, but UDT does not extend OBJECT*
- *221 Method declared ABSTRACT, but UDT does not extend OBJECT*
- *222 Not overriding any virtual method*
- *223 Implemented body for an ABSTRACT method*
- *224 Override has different return type than overridden method*
- *225 Override has different calling convention than overridden method*
- *226 Implicit destructor override would have different calling convention*
- *227 Implicit LET operator override would have different calling convention*
- *228 Override is not a CONST member like the overridden method*
- *229 Override is a CONST member, but the overridden method is*

not

- *230 Override has different parameters than overridden method*
- *231 This operator cannot be STATIC*
- *232 This operator is implicitly STATIC and cannot be VIRTUAL or ABSTRACT*
- *233 This operator is implicitly STATIC and cannot be CONST*
- *234 Parameter must be an integer*
- *235 Parameter must be a pointer*
- *236 Expected initializer*
- *237 Fields cannot be named as keywords in TYPE's that contain member functions or in CLASS'es*
- *238 Illegal outside a FOR compound statement*
- *239 Illegal outside a DO compound statement*
- *240 Illegal outside a WHILE compound statement*
- *241 Illegal outside a SELECT compound statement*
- *242 Expected 'FOR'*
- *243 Expected 'DO'*
- *244 Expected 'WHILE'*
- *245 Expected 'SELECT'*
- *246 No outer FOR compound statement found*
- *247 No outer DO compound statement found*
- *248 No outer WHILE compound statement found*
- *249 No outer SELECT compound statement found*
- *250 Expected 'CONSTRUCTOR', 'DESTRUCTOR', 'DO', 'FOR', 'FUNCTION', 'OPERATOR', 'PROPERTY', 'SELECT', 'SUB' or 'WHILE'*
- *251 Expected 'DO', 'FOR' or 'WHILE'*
- *252 Illegal outside a SUB block*
- *253 Illegal outside a FUNCTION block*
- *254 Ambiguous symbol access, explicit scope resolution required*
- *255 An ENUM, TYPE or UNION cannot be empty*
- *256 ENUM's declared inside EXTERN .. END EXTERN blocks*

don't open new scopes

- *257 STATIC used on non-member procedure*
- *258 CONST used on non-member procedure*
- *259 ABSTRACT used on non-member procedure*
- *260 VIRTUAL used on non-member procedure*
- *261 Invalid initializer*
- *262 Objects with default [con|de]structors or methods are only allowed in the module level*
- *263 Static member variable in nested UDT (only allowed in toplevel UDTs)*
- *264 Symbol not a CLASS, ENUM, TYPE or UNION type*
- *265 Too many elements*
- *266 Only data members supported*
- *267 UNIONS are not allowed*
- *268 Arrays are not allowed*
- *269 COMMON variables cannot be object instances of CLASS/TYPE's with cons/destructors*
- *270 Cloning operators (LET, Copy constructors) can't take a byval arg of the parent's type*
- *271 Local symbols can't be referenced*
- *272 Expected 'PTR' or 'POINTER'*
- *273 Too many levels of pointer indirection*
- *274 Dynamic arrays can't be const*
- *275 Const UDT cannot invoke non-const method*
- *276 Elements must be empty for strings and arrays*
- *277 GOSUB disabled, use 'OPTION GOSUB' to enable*
- *278 Invalid -lang*
- *279 Can't use ANY as initializer in array with ellipsis bound*
- *280 Must have initializer with array with ellipsis bound*
- *281 Can't use ... as lower bound*
- *282 FOR/NEXT variable name mismatch*
- *283 Selected option requires an SSE FPU mode*

- 284 Expected relational operator (=, >, <, <>, <=, >=)
- 285 Unsupported statement in -gen gcc mode
- 286 Too many labels
- 287 Unsupported function
- 288 Expected sub
- 289 Expected '#ENDIF'
- 290 Resource file given for target system that does not support them
- 291 -o <file> option without corresponding input file
- 292 Not extending a TYPE/UNION (a TYPE/UNION can only extend other TYPES/UNIONS)
- 293 Illegal outside a CLASS, TYPE or UNION method
- 294 CLASS, TYPE or UNION not derived
- 295 CLASS, TYPE or UNION has no constructor
- 296 Symbol type has no Run-Time Type Info (RTTI)
- 297 Types have no hierarchical relation
- 298 Expected a CLASS, TYPE or UNION symbol type
- 299 Casting derived UDT pointer from incompatible pointer type
- 300 Casting derived UDT pointer from unrelated UDT pointer type
- 301 Casting derived UDT pointer to incompatible pointer type
- 302 Casting derived UDT pointer to unrelated UDT pointer type
- 303 ALIAS name string is empty
- 304 LIB name string is empty
- 305 UDT has unimplemented abstract methods
- 306 Non-virtual call to ABSTRACT method
- 307 #ASSERT condition failed
- 308 Expected '>'
- 309 Invalid size
- 310 ALIAS name here is different from ALIAS given in DECLARE prototype
- 311 vararg parameters are only allowed in CDECL procedures
- 312 the first parameter in a procedure may not be vararg

- *313 CONST used on constructor (not needed)*
- *314 CONST used on destructor (not needed)*
- *315 Byref function result not set*
- *316 Function result assignment outside of the function*
- *317 Type mismatch in byref function result assignment*
- *318 -asm att\intel option given, but not supported for this target (only x86 or x86_64)*
- *319 Reference not initialized*
- *320 Incompatible reference initializer*
- *321 Array of references - not supported yet*
- *322 Invalid CASE range, start value is greater than the end value*
- *323 Fixed-length string combined with BYREF (not supported)*

Third party programs errors

These errors occur after the source has been compiled into assembler, they come from the auxiliary programs FB requires to compile a source into an executable: the linker, the assembler and (for Windows programs) the resource compiler.

If an IDE or a make utility are been used, additional errors can arise. These errors are outside the scope of this help.

External tools the FreeBASIC compiler (fbc) may invoke during the compilation process.

Description

FreeBASIC uses several tools for compiling source code in addition to the fbc compiler. The exact tools used by fbc and how they are invoked depends on how fbc was configured, the host platform (where fbc is running), the target platform (where the produced executable will be run), and other options (like environment variables and command line options).

FreeBASIC (fbc) may have been configured in one of two ways: either as standalone or prefixed. The standalone version searches directories relative to where the executable is located. The prefixed version has a hardcoded path configured in to the compiler indicating where it expects to find additional tools and libraries. For more information on configuring FreeBASIC, see the `INSTALL` text file located in the `src/compiler` directory of the FreeBASIC sources.

You can check if your installed version of fbc is "standalone" or "prefixed" by invoking fbc with the ***-version*** command line option.

Standalone

If fbc was configured as "standalone", it will search for files relative to where the fbc executable is located. fbc is at the "top" of the directory tree and searches sub-directories below it. The "top" directory (which defaults to the location where fbc is located) can be overridden with the ***-prefix*** command line option. "topdir" shown in the directories below represents the directory where the fbc executable is located, or the directory specified with the ***-prefix*** command line option (if it was given). "<target>" refers to the target platform having the same name as specified by the ***-target*** option.

If not cross compiling, fbc looks in these locations:

- /topdir/inc

- /topdir/lib/<target>
- /topdir/bin/<target>
- gcc is queried for missing libraries (currently on linux/freebsd only)

If cross compiling, fbc looks in the these locations:

- /topdir/inc
- /topdir/lib/<target>
- /topdir/bin/<target>
- gcc is not queried (only target library directory is used)

Prefixed

If fbc was configured as "prefixed", it will search for files relative to the configured prefix (hardcoded in the fbc executable). "prefix" shown in the directories below represents the configured prefix, or the directory specified with the **-prefix** command line option (if it was given). "<target>" refers to the target platform having the same name as specified by the **-target** option.

If not cross compiling, fbc looks in these locations:

- /prefix/include/freebasic
- /prefix/lib/freebasic/<target>
- /prefix/bin/freebasic/<target>
- gcc is queried for missing libraries (currently on linux/freebsd only)

If cross compiling, fbc looks in the these locations:

- /prefix/include/freebasic
- /prefix/lib/freebasic/<target>
- /prefix/bin/freebasic/<target>
- gcc is not queried (only target library directory is used)

GCC Queries

If fbc is unable to locate a file, it may invoke `gcc -print-file-name=<file>` to query the location of the file. The following are files that may

be located using gcc:

- crt1.o
- crtbegin.o
- crtend.o
- crti.o
- crtn.o
- gcrt1.o
- libgcc.a
- libsupc++.a
- libc.so (Linux only)

Finding Binaries

fbcc will invoke additional tools (binary executables) as part of the compiling and linking process. The following is a list of tools (executables) that may be invoked by fbcc depending on the host platform, target, or type of executable or library to be produced:

- as
- ar
- ld
- gcc
- GoRC
- dlltool
- pexports
- cxbe

fbcc will search for these tools in the following manner:

- If an environment variable (having same name as the tool without any extension, all in uppercase) has been set, it explicitly indicates the path and name of the executable to be invoked.
- If the file (or a symlink) exists in `prefix/bin/freebasic/<target>`, or `./bin/<target>` for the standalone version, then use it.
- On Linux, if the tool could not be found in

prefix/bin/freebasic/<target>, Or ./bin/<target> for the standalone version, fbc tries to invoke it anyway as may be installed on the system and located on the PATH

"<target>" refers to the target platform having the same name as specified by the *-target* option.

See also

- **Running FreeBASIC**
- **Compiler Command Line Options**
- **Compiler FAQ**

FreeBASIC the Successor

FreeBASIC is designed as an official successor of sorts to a high level compiler for MS-DOS titled "QuickBASIC", which compiled BASIC code, an easy-to-read programming language created in 1964 by John Kemer and Thomas Kurtz. "QB" was packaged with a user-friendly IDE and interpreter that made it very easy to write custom applications. This line of products is officially continued today in the form of "Visual Basic", part of Microsoft's Visual Studio .NET programming suite.

Microsoft and BASIC Products

Microsoft and BASIC extend far prior to QuickBASIC. In fact, Microsoft's first product was a small BASIC interpreter for Altair computers released in 1975, and until the early 1980s Microsoft was known only as a language vendor. They ported their BASIC software to several different personal computers at the time and made decent business doing it.

In August of 1981 Microsoft released the next major step in its BASIC line, "Advanced BASIC", as part of a commission for IBM's PC-DOS, and is more often called by its executable name, BASICA.EXE. For Microsoft's new MS-DOS, they released GW-BASIC, which was, for the most part, a port of BASICA that did not require IBM's Basic ROM included with its systems.

BASICA and GW-BASIC are interpreters. Interpreters read source code and "interpret" it into computer code as it is read. This is useful, but slow. Microsoft, in 1983, released BASCOM for MS-DOS. BASCOM compiled BASIC code into native machine code, which ran much faster than interpreted code. This was repackaged with an IDE and released as QuickBASIC in 1985.

QuickBASIC

From 1985 to 1992, QuickBASIC was the primary BASIC product, released by Microsoft and using BASCOM, and later the Microsoft BASIC Compiler. In 1991, a slimmed down interpreter often thought to be the

missing "QuickBASIC 5.0" was packaged with MS-DOS 5.0 and released as "QBasic 1.1".

QuickBASIC as a BASIC dialect provides a loose standard for modern BASIC compilers. It abolishes the need for line numbers as used in previous BASIC interpreters, is case sensitive and has keywords that are in plain English. QuickBASIC also featured a runtime library, a library compiled by default and usable in source code, with many useful commands.

In 1991, Microsoft combined a drag-and-drop GUI designer made in 1988 called 'Ruby' with QuickBASIC. This product was called "Visual Basic", and marks the beginning of the end of QuickBASIC. Microsoft released one last version of QuickBASIC called "Visual Basic for DOS" in 1992, and discontinued the product forever.

The Internet and QBasic's Second Wind

Because the "QBasic 1.1" interpreter was packaged with MS-DOS, it was released with every copy of DOS until its dying days, Windows 3.1, and even Windows 95, 98 and ME. With the wild success of Windows, QBasic became the most widely available programming tool available for Microsoft operating systems.

When the World Wide Web became popular in the mid-90s, many hobbyist programmers made websites dedicated to QuickBASIC not as an application tool, but as a platform for their demos and games. Many assembly libraries were created for it after Microsoft dropped support, and as these demos and games became more elaborate, so did the "QBasic Community". From the mid-90s, through the new millennium to today, QuickBASIC has enjoyed a small but present cult following.

Andre Victor, FreeBASIC's creator, was first known over the internet as the author of several extensions to QuickBASIC in the form of libraries. He created routines to improve the speed of floating point operations, access the internet, use SVGA graphics, and provide powerful QBasic language programming features. In the late summer of 2004, he began work on a 32-bit compiler using Visual Basic for DOS.

FreeBASIC is Born

FreeBASIC was first programmed in VB-DOS, with the goal of compiling itself. Because of this, both its syntax and runtime library are designed to emulate QB's syntax and runtime as far as it is practical in a 32-bit Windows environment. For the most part, the two dialects are extremely similar, and most code can be ported with little or no modification, though in some cases routines reliant on 16-bit DOS must be rewritten. The resulting compiler shares a greater similarity to QB than any compiler on the market, including Visual Basic.

Because of its open source, its well-written code and its similarity to QB, FreeBASIC has become popular among the "QB Community" and its boundaries continue to grow as it receives more attention and gathers more features that promise to move BASIC into the future.

Since version 0.17, FreeBASIC introduced a `-lang` command-line option that is used to change the language compatibility mode. Use the `-lang qb` option when compiling to select the most QB compatible parser. All differences listed below assume that `-lang qb` was used.

Architecture/Platform incompatibilities

- FreeBASIC is written for 32-bit operating systems and a 32 bit DOS extender, and cannot utilize code which depends on 16-bit DOS, 16 bit assembly or memory model (segment & offset, XMS/EMS, ...).
- `DEF SEG` is no longer necessary and will not work - any code which `POKES` to video memory this way will no longer function, however, for DOS it can be easily rewritten using DPML features.
- `CALL INTERRUPT` no longer functions, as it relies on 16-bit DOS. DOS interrupts can be called in the DOS32 version by using the DPML library, but they might work slowly because of the 32bit-16bit-32bit mode changes the processor will have to perform.

Changed due to ambiguity

- A scalar variable and an array with the same name and suffix can no longer share the same name.
- `SHARED` can't be used inside a `SUB` or `FUNCTION` as it resulted in shared variables not defined in the main program. A proper `DIM SHARED` in the main program must be used.
- `COMMON` declarations do not depend on the order they are made, variables are matched by name and for that reason named `COMMON` is no longer supported. All `COMMON` arrays are variable-length arrays in FB.
- If a single line `if` has an (unnecessary) colon directly after the `THEN`, it has to be terminated by an `End If` in FB. If that unneeded colon is removed, FB will behave as QB.

Design differences

- Graphics support was internally redesigned, see [GfxLib overview](#)

- `CLEAR` is no longer used to reset all variables and set the stack. Variables must be reset one by one, and stack size can be changed in the compiler command line. The keyword `CLEAR` is used to do memory fills in FB.
- String `DATA` items must be enclosed in quotes in FB, in QB this was optional
- All functions must have been declared, even with a `CALL` in FreeBASIC. With `CALL` it was possible to invoke prototype-less functions in QuickBASIC. (to be supported in future with `-lang qb`)
- In FreeBASIC all arrays must be explicitly declared. (Interpreted QuickBASIC arrays are automatically created with up to 10 indices.)
- Strings use a null (char 0) terminator to be compatible with C libraries and the Windows API, fixed-length strings can't contain `chr$(0)` chars for now.
- When `INKEY[$]` reads an extended key (Num Pad, Arrows...) it returns a two character string. In FB the first character is `CHR[$]` (255), while in QB this first char is `CHR$(0)`.
- With fixed length strings FreeBASIC gives the real length as `Len` plus one (null-char), even on `Type` fields.
- In FreeBASIC, unused characters of fixed-length strings are set to 0, regardless of what `"-lang"` compiler option is used. In QB, unused characters are set to 32 (space, or " ", in ASCII).
- When a fixed-length string is declared, but still not initialized, all characters are set to 0, both in FreeBASIC and QB.
- The arrays are stored in row-major order in FB, they were stored in column-major order in QB by default. Row major order: data with the same last index are contiguous. Column-major order: data with the same first index are contiguous. For example, if you have `DIM A(1 TO 3, 1 TO 8)`, in row-major order the elements are stored such that `A(3,5)` is followed in memory by `A(3,6)`; in column-major order `A(3,5)` is followed in memory by `A(4,5)`.
- Programs don't stop anymore on runtime errors unless `-e` or `-ex` option is used in the command line. Using these options allow the use of QB style error handling (`ON ERROR, RESUME...`).
- Octal numbers are written `&o...;`, whereas in QB they could be

written as `&o...;` or `&....`

- In FB FOR loops in subs/functions do not accept arguments received byref as counters. A local variable must be used.
- FB's **Locate** does not respect the fourth and fifth arguments for cursor shape.
- FB's **Screen** does not allow switching the visible or the work-page. Use **ScreenSet** instead.
- FB's **Color** does not allow a third argument for border color.
- FB's **Timer** returns the number of seconds since the computer started, while QB's **TIMER** returns the number of seconds since midnight. (Win32 and Linux only: No more wrapping at midnight! :))
- In QB a `chr$(13)` inside a string did a CR+LF when PRINTED. In FB the `CHR(13)` prints just at what it is, a CR.
- EOF can no longer be used to detect an empty comms buffer. Empty buffer should be tested comparing `LOC` with `0` in FB. Also, for files opened in `RANDOM` or `BINARY` mode, EOF returns non-zero already after reading exactly the file size, see **EOF**.
- Integer variables do not signal overflow errors in FB, even with the **-ex** option on. Any QB code relying in catching integer overflow errors will not work in FB.

Archaic commands

- **BSAVE** and **BLOAD** can be used in FB only to save and retrieve screens or graphic buffers. They will work only if `gfxlib` is linked, this is, if a graphics screen mode is requested somewhere in the program. The console can't be saved with **BSAVE** or retrieved with **BLOAD**. The other use of **BSAVE-BLOAD**, saving and loading full array can be achieved easily with **GET** and **PUT**.
- **FIELD** statement (for record definition at runtime) has been left aside. The keyword **FIELD** is used in FB to specify field alignment in **Type** variables.
- **PC Speaker** commands no longer function: Any references to **SOUND** or **PLAY** statements will result in an error message. There is a third party library available to emulate this functionality, but it's not included with FreeBASIC.

- Fake event-driven programming (ON KEY, ON PEN, ON STRIG, ON TIMER) no longer works. They could be emulated by a separate library.
- MKSMBF\$ and all the MKxMBF\$ commands supporting the pre-QB4.0 Microsoft proprietary floating point format (MBF) are not implemented.
- The use of parenthesis in the arguments passed to a function to emulate by-value passing is not permitted. The CALL quirk resulting in all arguments being passed by value, no longer works. The proper **ByVal** and **ByRef** keywords must be used.
- FILES is not implemented. Instead, PDS 7.1-compatible **Dir[\$]** can be used.
- IOCTL, ERRDEV and ERRDEV\$, low level functions to access hardware are not implemented as they were OS-dependent.
- CALL ABSOLUTE to run inline machine code is no longer supported. Instead you can use **Asm** . . . END ASM blocks to insert inline assembler commands. Or use the ASM . . . one line command.

FreeBASIC Dialects



FreeBASIC version 0.17b introduces a **-lang** command-line option, used to change the language compatibility mode for different dialects of the basic language.

Starting with version 0.18.3b the **-lang qb** dialect has been further restricted to only allow what would have been allowed in **QuickBASIC**.

In version 0.18.4b the **-lang fblite** dialect was added, intended to replace **-lang deprecated** in future.

In version 0.20.0b the **#lang** directive and **\$Lang** metacommand were added to specify a dialect from source.

-lang option	description
fb	FreeBASIC compatibility (default)
qb	qbasic compatibility
fblite	FreeBASIC language compatibility, with a more QBASIC-compatible coding style
deprecated	compatibility with FB 0.16

The **-lang** option was needed to allow FreeBASIC to support object-orientation and other features in the future, without crippling the QuickBASIC support or breaking compatibility with old FreeBASIC sources, and without making FreeBASIC difficult to maintain with many different versions of very similar packages. The QuickBASIC support can continue to be improved, if needed, without breaking the sources written specifically for FreeBASIC.

To compile old GW-BASIC or QuickBASIC/QBasic sources without too many changes, use the **-lang qb** option on the command-line when running `fbcc`. This option will evolve into a better compatibility with QuickBASIC/QBasic code.

To compile FreeBASIC sources from 0.16b, use the **-lang deprecated** option. This option is maintained for compatibility and will not evolve in

the future, and it's likely to disappear when FreeBASIC reaches a non-beta release.

For programmers who want to access some of FreeBASIC's newer features, but want to retain a more QBASIC-friendly programming style, use the **-lang fblite** option. This dialect will not undergo significant changes in the future, but will continue to be maintained and supported. This option is also the most compatible with sources that were made using older versions of FreeBASIC.

It is recommended to use **-lang fb** for new projects, as new features (object classes, inheritance..) will be added exclusively to this dialect.

-lang fb (the default mode)

Not supported:

1) implicit variable declaration

- All variables must be explicitly declared, using **Dim**, **ReDim**, **Var**, **Const**, **Extern** Or **Common**.

2) type suffixes (!, #, \$, %, &)

- They are only allowed for numeric literals, but it's recommended to use **cast** or the **f** (single), **d** (double), **ll** (longint), **ul** (ulong), **ull** (ulongint) numeric literal suffixes to resolve overloading.

3) **DefByte**, **DefUByte**, **DefShort**, **DefUShort**, **DefInt**, **DefUInt**, **DefLng**, **Deflongint**, **Defulongint**, **DefSng**, **DefDb1**, **DefStr**

- An explicit type ("As T") is needed when declaring variable using **Dim**, **ReDim**, **Extern** or **Common**. Variables declared using **Var** or **Const** have their types inferred from an initialization value (an explicit type is optional using **Const**).

4) all parameters passed by reference by default

- By default, all intrinsic scalar types - numeric and pointer types - are passed by value (**ByVal**). Any other type - **Stri** or user-defined type - is passed by reference (**ByRef**).

- Use the `-w pedantic` command-line option to have parameters without explicit `ByVal` or `ByRef` reported.

5) OPTIONS of any kind (no context-sensitivity)

- Instead of `Option NoKeyword`, use `#undef`.
- Instead of `Option Escape`, use: `!"some esc seq \n\r"` (notice the `!` char) and pass `-w pedantic` to check for possible escape sequences.
- `Option Explicit` isn't needed, see item 1.
- Instead of `Option Dynamic`, declare variable-length arrays using `ReDim`. `Dim` can also be used to declare variable-length arrays using variable or no subscripts.
- Instead of `Option Base`, use explicit lower-bound subscript in arrays declarations.
- Instead of `Option Private`, use `Private` to declare procedures with internal linkage.
- Instead of `Option Gosub` and `Option Nogosub`, use procedures with `Sub` or `Function`.

6) periods in symbol names

- Use namespaces instead.

7) `GoSub` OR `Return (From Gosub)`

- Nested procedures may be allowed in future.

8) `On Gosub` OR `On Goto`

- Use `Select Case As Const` *expr* for the latter.

9) `Resume`

- Most runtime and graphics library procedures now return an error code, like:

```
IF OPEN( "text" FOR INPUT AS #1 ) <> 0 THEN error...
```

10) `'$DYNAMIC`, `'$STATIC`, `'$INCLUDE` meta-commands embedded in comments

- See item 5 about `Option Dynamic`.

- Use `#include "filename"` instead of `'$include`.

11) `call` or `let`

- Just remove them.

12) numeric labels

- Named labels can be used instead, e.g. `label_name: / goto label_name`.

13) global symbols with the same name as keywords

- Declare them inside a namespace.

-lang deprecated

Supported: *Anything allowed in version 0.16b, but:*

1) `GOSUB/RETURN` and `ON ... GOSUB` (even at module-level)

- so the `GOSUB` implementation could be thread-unsafe in the `-lang qb` mode, allowing fast execution (`-lang qb` doesn't support multi-threading, while `-lang deprecated` does).

Not supported:

1) Classes

- Periods allowed in symbol names make it too difficult and/or ambiguous.

2) Operator overloading

- Periods allowed in symbol names make it too difficult and/or ambiguous.

3) Constructors, destructors and methods in `TYPES`.

- Periods allowed in symbol names make it too difficult and/or ambiguous.

-lang fblite

Supported: *Anything allowed in the **-lang deprecated** dialect, plus..*

1) GOSUB/RETURN

- Use **option Gosub** to enable. This will disable RETURN from exiting a procedure, due to ambiguity.

Not supported:

1) **scope** blocks

- All variables are given procedure scope. Explicit **Scope** blocks may be added later.

-lang qb

Supported: *Everything not supported/allowed in the **-lang fb** dialect, plus..*

1) **call** can be used with forward-referenced functions.

2) **shared** can be used inside functions. (W.I.P.)

3) All variables, implicitly or explicitly declared, are always allocated in the procedure scope, like in QuickBASIC.

4) The **data** statement won't look up symbols, every token is assumed to be a literal string even without quotes, like in QuickBASIC.

Not supported:

1) Multi-threading

- None of the **threading** procedures may be used.

2) Classes and Namespaces

3) Procedure and operator overloading

4) Constructors, destructors and other member procedures in **Type** definitions.

5) **Scope** blocks

6) **Extern** blocks

7) Variable initialization

- All variables are moved to the procedure scope (like in QuickBASIC), making initializing local variables too difficult to support.

FreeBASIC questions:

- What is FreeBASIC?
- Who is responsible for FreeBASIC?
- Why should I use FreeBASIC rather than QBasic?
- Why should I use FreeBASIC rather than some other newer BASIC ?
- How fast is FreeBASIC?
- How compatible is FreeBASIC with QuickBASIC?
- How compatible is FreeBASIC with Windows? DOS? Linux?
- Does FreeBASIC support Object Oriented Programming?
- What are the future plans with FB / ToDo list ?
- Can I program GUI applications in FB ?
- Is FB suitable for complex / big applications?
- Can I use a non-latin charset in my FreeBASIC applications?
- Can I use Serial/COM and Hardware/CPU ports in FB?

Getting Started with FreeBASIC questions

- Where can I find more information about FreeBASIC?
- Why doesn't the QB GUI open when I start FreeBASIC?
- Can I have an offline version of the documentation?
- What's the idea behind the FB dialects?
- Why does my program crash when I define an array larger than xx?
- Why does my program fail to compile with the message 'cannot find -llibname'?

Advanced FreeBASIC

- How do I link to C libraries?
- Can I use a debugger?
- What's the goal of the AR.EXE, AS.EXE and LD.EXE files included with FB ?
- Is there a limit on how big my source files can be?
- Can I write an OS in FreeBASIC ?

- I'm developing an OS, can FreeBASIC be ported to my OS ?
- Does FreeBASIC support returning reference from Functions, like in C++?

See also

FreeBASIC questions

What is FreeBASIC?

FreeBASIC is a free, 32-bit BASIC compiler for Windows (32-bit), 32 bit protected-mode DOS (COFF executables, like DJGPP), and Linux (x86) It began as an attempt to create a code-compatible, free alternative to Microsoft QuickBASIC, but has quickly grown into a powerful development tool, already including support for libraries such as Allegro, SDL, OpenGL, and many others with its default installation.

Aside from having a syntax mostly compatible with QuickBASIC, FreeBASIC introduces several new features to the aged language, including pointers to variables and functions, and unsigned data types.

FreeBASIC compiler is self-hosting - written in FreeBASIC, the libraries however are written in C.

[Back to top](#)

Who is responsible for FreeBASIC?

The first versions of FreeBASIC were developed exclusively by V1ctor. Later versions gained contributions from many people, including Lillo, who developed the Linux port and the graphics library, and DrV, who developed the DOS port.

See the [FreeBASIC Credits](#) page.

[Back to top](#)

Why should I use FreeBASIC rather than QBasic?

FreeBASIC has innumerable advantages over QBasic, QuickBASIC, PDS, and Visual Basic for DOS.

- It supports 32-bit processors, where QBasic is designed for 16-bit CPU's.
- It supports modern OSes. It has ports to Windows, Linux, and 32-bit DOS.
- It supports modern APIs such as SDL, DirectX, Win32, and OpenGL.
- It is distributed under the GPL, meaning it's free and legal to use, unlike most copies of QuickBASIC / other BASICs.
- The library is distributed under the LGPL with additional exception, meaning you may do whatever you want with your compiled programs, including selling them.
- FreeBASIC is many times faster than QuickBASIC / other BASICs.
- FreeBASIC supports many features, such as pointers and inline Assembly, which are not available in QuickBASIC / other BASICs.
- QuickBASIC only supports DOS. Windows support for DOS emulation (and thus QuickBASIC) is becoming thinner with every new version. Vista does not support graphics or fullscreen text for DOS applications.

[Back to top](#)

Why should I use FreeBASIC rather than some other newer BASIC

FreeBASIC has many traits which make it more desirable than most other BASIC language implementations:

- FreeBASIC adheres closely to the standard BASIC syntax making it easier to use.
- FreeBASIC is compiled to actual programs (executables), not bytecode.
- FreeBASIC has a large, dedicated community which has actively participated in the development of FreeBASIC.
- FreeBASIC utilizes standard methods of accessing

common C libraries. SDL, for example, is standard C SDL not a new set of intrinsic commands.

- FreeBASIC has ports to Windows, Linux, and 32-bit DOS. It retains consistent syntax between the three ports.

[Back to top](#)

How fast is FreeBASIC?

Most tests run by the community have shown FreeBASIC is significantly faster than QuickBASIC, faster than most other GPL or commercial BASICs, and often approaching GCC in terms of speed.

The [Computer Languages Benchmark Game](#), an independent test team, give FreeBASIC for Linux a speed 1.8 times slower than GNU g++. Tests are about calculation, memory and disk access speed in console programs, no graphics capabilities were tested. This is not a bad result considering FreeBASIC is not yet an optimizing compiler.

One area where there is a notable speed deficiency is in 32-bit console modes. While FreeBASIC is consistently on-par with other 32-bit console mode applications, 32-bit console mode operations are significantly slower than 16-bit console operations, as seen in QuickBASIC. In DOS version, some I/O operations can slow down after porting from a 16-bit BASIC to FB - optimizing the code brings the speed back.

[Back to top](#)

How compatible is FreeBASIC with QuickBASIC?

The FreeBASIC built in graphics library emulates the most used QB graphics modes (modes 7,12,13) and implements all the drawing primitives featured in QB.

Most compatibility problems arise from the use of 8086-DOS-hardware specific low-level techniques in the old QB programs. VGA port programming, DOS interrupts, memory segment switching, poking to the screen memory or music playing using the PC speaker are not directly supported, even if they can be supported/emulated by external libraries. Other issues in porting old QB programs, like variable name clashes with new FB keywords, variables with the name of a QB keyword plus a type suffix, default integer size being 32 bits in FB, are addressed by running FreeBASIC with the commandline switch `-lang qb`.

See [Differences between FreeBASIC and QuickBASIC](#).

[Back to top](#)

How compatible is FreeBASIC with Windows? DOS? Linux?

FreeBASIC is fully compatible with Windows, MS-DOS, FreeDOS and Linux. When planning to create a program for all three platforms, however, keep API availability in mind -- code utilizing OpenGL will work in Windows and Linux, for example, but won't in DOS, because OpenGL is not available for DOS.

[Back to top](#)

Does FreeBASIC support Object Oriented Programming?

FreeBASIC (since version 0.90) supports classes (user-defined types) with member functions (methods), static methods, static member variables, constructors, destructors, properties, operator overloading, single inheritance, virtual and abstract methods (polymorphism) and run time type information. Future plans regarding OOP functionality include adding support for multiple inheritance and/or interfaces. For more information see: [A Beginners Guide to Types as Objects](#).

[Back to top](#)

What are the future plans with FB / ToDo list ?

You can find out what's planned for the future releases by directly looking at [fbc's todo.txt](#).

[Back to top](#)

Can I program GUI applications in FB ?

Yes, you can. Headers allowing you to call the GUI API of Windows and Linux are supplied with the respective versions, but the programs made this way are not portable.

There are some API wrappers and experimental RAD applications that create non-portable GUI code for Windows.

For portable programming a multiplatform GUI wrapper library as GTK o

wx-Widgets may be used. GTK headers are provided with FB, but the OOP functionality currently available in FB prevents the use of wx-Widgets. The programs created with these libraries may require the use to install the wrapper libraries in their systems.

For games and small graphics applications there are some FB-specific libraries that draw and manage simple controls as buttons and edit boxes inside the graphics screen, programs made with those libs are entirely portable.

[Back to top](#)

Is FB suitable for complex / big applications?

The FB compiler is self-hosting, it is programmed itself in FB. That means more than 120 000 lines of code at the moment, a fairly complex application.

[Back to top](#)

Can I use a non-latin charset in my FreeBASIC applications?

FreeBASIC has the Unicode support provided by the C runtime library for the given platform. This means FB DOS won't help you with Unicode. On other platforms you can use **Wstrings** to support any charset you need. The File OPEN keyword has an additional **Encoding** parameter allowing for different encodings. As FreeBASIC is coded itself in FB, this means you can code your source in an Unicode editor so the comments and string literals can be in any character set (keywords, labels and names for variables and procedures must be kept inside the ASCII set..).

For the output to screen the support is different from console to graphics. In console mode wstring printing in non latin charsets is supported if the console font supports them. Graphics mode uses an internal CP437 charset (the old DOS charset) font so non-latin output requires a custom made raster font and the use of the **Draw String** keyword. Third party tools exist to grab an external font and convert it to the DRAW STRING format.

[Back to top](#)

Can I use Serial/COM and Hardware/CPU ports in FB?

Yes, FB has built in functions to access the serial/COM port and hardware/CPU ports with no need of external libraries. See the **OS specific FAQ's** for details for your OS, and **Open Com, Inp** and **Out** .

Back to top

Getting Started with FreeBASIC questions

Where can I find more information about FreeBASIC?

The FreeBASIC Wiki is the most up-to-date manual for using FreeBASIC available **here**.

Active FreeBASIC related forums, besides the **official one**, can be found at **qbasicnews**, **Pete's QB Site** , **the FB Games directory** or **freebasic portal.de (in German)**.

Active magazines which regularly have FreeBASIC related articles are **QB Express** and **QBXL Magazine**. These magazines are always looking for new articles, so if you think you've got a good idea for an article about FreeBASIC, submit it!

Back to top

Why doesn't the QB GUI open when I start FreeBASIC?

QB had an Integrated Development Environment (IDE). FreeBASIC does not.

FreeBASIC is only a compiler, not a complete QuickBASIC clone. It is a console mode application. It will accept a BAS file on the command line, and spit out an EXE file.

You can create the BAS file with the simplest plain text editor in your OS (Notepad, EDIT, nano,...), then run the compiler.

If you can't live without syntax coloring, error highlighting, multiple file managing, integrated debugger, context help or other features, you need an IDE. See the **OS specific FAQ's** for the IDE's and editors available.

Back to top

Can I have an offline version of the documentation?

This online Wiki is the official documentation for FB. Usually it is up-to-date with the latest improvements found in the development version of FB.

Offline versions of this wiki (in CHM, HTML and other formats) are available from the [Documentation directory at fbc's downloads site on SourceForge](#).

[Back to top](#)

What's the idea behind the FB dialects?

The idea is to allow improvements in the language while maintaining backwards compatibility with QB code. The quirks of the QB syntax are not compatible with the more rigid style required by OOP. The new FB keywords often clashed with variable names in old QB programs. QB allowed to use freely dots in variable names and procedures not being UDT's.

The three dialects (-lang fb, -lang qb, -lang fblite) allow to combine the best of two worlds.

- lang fb provides the framework required for OOP programming . Other dialects don't give access to OOP.
- lang qb will allow the developers to keep increasing the compatibility with qb programs. Newer keywords in FB can be used by preceding them with two underscores. For example, Getmouse can be called by using __Getmouse
- lang fblite offers FreeBASIC language compatibility, with a mor QBASIC-compatible coding style.

See [Compiler Dialects](#) for details.

[Back to top](#)

Why does my program crash when I define an array larger than xx '

This generally happens because you made an *automatic* fixed-length array too large, and it is corrupting the program stack. You have a couple of options:

- if possible, reduce the size of the *automatic* array

- create a variable-length array, by
 - defining the array with an empty subscript list (using `Dim`), or
 - defining the array with variable subscripts instead of numeric literals, `Constants` or `Enums` (using `Dim`), or
 - defining the array with `ReDim`
- reserve more memory for the program stack by using the `-t` **command-line option** when compiling. The default is `-t 1024` (kilobytes). Note: it's a bad idea to use very large values here.
- create a *static* array by defining the array with `Static` rather than `Dim` (only locally visible, but globally preserved)
- define the array with `Shared` access using `Dim` (this makes the array fully global)
- use **Pointers** and **Memory Functions** like `Allocate` and `Deallocate` to manage memory yourself - this is the preferred way for storing big buffers, but not for beginners.

Static and *variable-length* arrays don't use the program stack for their element data, so do not have the problem associated with *automatic* fixed-length arrays. See **Storage Classes** for more information. Note that storing huge buffers as *static* or increasing the stack size far above the default is not a very good idea, since it increases the fixed amount of memory needed to load and start your program, even if most of it is not used later, and can result in performance degrade, or even refusing your program to load at all.

Why does my program fail to compile with the message 'cannot find -llibname'?

This is an error raised by the linker. The program is supposed to link to an external library, designated in the program code with `#inclib` or on the compiler command line with `-l`. However, the linker has been unable to find a matching file in any of the library paths. Check the homepage of the library you want to compile with to find out how to download it, or check ExtLibTOC to see if information about the library can be found there.

[Back to top](#)

Advanced FreeBASIC

How do I link to C libraries?

C libraries are set up in much the same way in FreeBASIC as they are in C. Every library included with FreeBASIC has a basic include file named "*library name.bi*" which uses the **#inlib** metacommand to include the library, and the **Declare Statement** to declare the functions within the library. FreeBASIC includes hundreds of BI files, see full list of library headers [here](#).

[Back to top](#)

Can I use a debugger?

FreeBASIC can use preferably a debugger compatible with GNU GDB.

- Win32: Insight is an user friendly wrapper for GDB, see [Win32 related FAQ](#).
- DOS: Be warned that DOS also has product named "Insight", but it's a real mode debugger not usable with FreeBASIC, use GDB or some DPMS32 debugger at least.
- Linux: use GDB.

See the [OS specific FAQ's](#) for details for your OS.

[Back to top](#)

What's the goal of the AR.EXE, AS.EXE and LD.EXE files included with FB ?

AS.EXE is GAS, the "GNU assembler". It is always involved in compilation. LD.EXE is the "GNU linker", involved in creation of executables. AR.EXE is the "GNU archiver", in fact a librarian, creating libraries.

[Back to top](#)

Is there a limit on how big my source files can be?

Yes, since FreeBASIC is a fully 32-bit compiler it may operate on source files up to theoretically 4GB or 4294967296 bytes, however your RAM capacity should be significantly above the size of your source, otherwise the compilation won't finish or will be very slow at least.

[Back to top](#)

Can I write an OS in FreeBASIC ?

YES and NO. If you really insist to write an OS and involve FB, the answer is YES. If the question is, whether it is a good idea that you, even more if a beginner, should start coding an OS using FB now, the answer is NO. Several pitfalls apply:

- OS development is hard, see http://www.osdev.org/wiki/Getting_Started .
- FB won't help you to bypass the need to deal with assembly, also C might be almost impossible to avoid.
- You won't be able to use most of the trusted FB features, like graphics, file I/O, threads, memory management, even console I/O ... just control flow, math and logic. If you need those library functions, you will have to **reimplement** them

FreeBASIC relies on GCC, and available informations about developing an OS in C apply to FreeBASIC as well. FB will help you neither more nor less than GCC.

[Back to top](#)

I'm developing an OS, can FreeBASIC be ported to my OS ?

Depends. If your OS at least equalizes the functionality of DOS with DPMI32 (console I/O (seeking, multiple files open, ...), file I/O, memory management) **and** has a port of GCC, then the answer is YES. If you have at least another somewhat compliant C compiler **with** libraries, it might be possible. You can't reasonably port FB for example to an OS allowing to load or save a file in one block only, or a 16-bit OS.

[Back to top](#)

Does FreeBASIC support returning references from Functions, like in C++?

Yes, this functionality exists since version 0.90.0. Procedures can now return references using **ByRef** as *datatype* for the return type.

[Back to top](#)

See also

- [Win32 related FAQ](#)
- [DOS related FAQ](#)
- [Linux related FAQ](#)

and

- [FB Runtime Library FAQ](#)
- [Frequently Asked FreeBASIC Graphics Library Questions](#)

Frequently Asked FreeBASIC Graphics Library Questions



FreeBASIC Graphics Library questions:

- How can I link/use Gfxlib?
- What about the fbgfx.bi header file?
- How are Get/Put arrays managed?
- Why is Bsave/Bload crashing?
- How can I get the red, green, blue, or alpha component of a colour?
- How can I make the 'x' button in the window header close my app?
- Can't run programs using Screen 13 or 14 in fullscreen !
- Why does Imagecreate return a NULL pointer?

FreeBASIC Graphics Library questions

How can I link/use Gfxlib?

Gfxlib is "built in" into the language, it is not necessary to include any .bi library explicitly. FreeBASIC detects you want to use Gfxlib when you use **ScreenRes** statements. So to use Gfxlib, just start a graphics screen mode with graphics commands.

[Back to top](#)

What about the fbgfx.bi header file?

The fbgfx.bi header file is available for inclusion by your program, and contains constants and type definitions that may be helpful to the programmer when using Gfxlib. You do not have to explicitly include this file to use Gfxlib however; the header is on hand as an aid. It contains the constants for the mode flags that can be passed to **ScreenRes**, and also definitions of **Keyboard scancodes** and the **fb.In** structure.

[Back to top](#)

How are Get/Put arrays managed?

In FreeBASIC, images can be used as arrays (as in QB) or as pointers. image data is contained in one continuous chunk. The chunk consists of by the image data. The header can be of two types (old-style and new-s the format of the following image data, for details see GfxInternalFormat

[Back to top](#)

Why is Bsave/Blod crashing?

Bsave/Blod can only be used to load and save graphics screens in Fre used to save a text mode screen. To load and save an array check this [Get/Put](#) .

[Back to top](#)

How can I get the red, green, blue, or alpha component of a color?

Each byte in a color attribute corresponds with the red, green, blue, and The following example shows how to extract the component values from color attribute.

```
#define rgb_a(x) ((x) Shr 24)
#define rgb_r(x) ((x) Shr 16 And 255)
#define rgb_g(x) ((x) Shr 8 And 255)
#define rgb_b(x) ((x) And 255)
```

```
Dim As UInteger c
Dim As Integer x, y
Dim As UByte red, green, blue, Alpha
```

```
' Assume a 16, 24, or 32 bit screen mode has been s
c = Point(x, y)
red = rgb_r(c)
green = rgb_g(c)
blue = rgb_b(c)
Alpha = rgb_a(c)
```

Back to top

How can I make the 'x' button in the window header close my applicati

In windowed graphics mode you can test for the press of the window's X **Inkey**, checking for the value `Chr(255) + "k"` (which is also the code returned by `Inkey`). This applies to Win32 and Linux, in DOS there is no "X" button.

Here is a small example:

```
' ' "X" close button example , Win32 and Linux only
Dim As String key
Screen 13
Do
    Print "Click the 'x' to close this app."
    Sleep
    key = Inkey
Loop Until key = Chr(27) Or key = Chr(255, 107) 'esc
button
```

Back to top

Can't run programs using Screen 13 or 14 in full-screen!

It's a hardware/driver limitation (Win32 and Linux only?). Video cards do not support low resolution graphic modes nowadays. If full-screen is required you should use at least Screen 17 or 18, or a resolution of 640x480 or higher to be sure the hardware can handle it.

Back to top

Why does Imagecreate return a NULL pointer?

ImageCreate needs to create an image buffer that fits the current screen mode. It cannot do so if there is no screen mode setup yet, so it returns NULL, and if you try to access a NULL pointer later on that crashes the program.

This is known to happen when Imagecreate is called *before* the graphics initialized with a call to **Screen** or **ScreenRes**, as may happen when Imagecreate is in a global constructor that is invoked before the Screen or Screenres call at the start of the program. In such a case it is necessary to move the screen initialization constructor too, and have it execute before the image-creating constructor.

[Back to top](#)

See also

- **[Compiler FAQ](#)**
- **[FB Runtime Library FAQ](#)**
- **[Frequently Asked FreeBASIC Graphics Library Questions](#)**

FreeBASIC Runtime Library questions:

- [How do I play sound?](#)
- [How do I access the serial ports?](#)
- [How do I print?](#)
- [How do I access the hardware ports?](#)

FreeBASIC Runtime Library questions

How do I play sound?

Of the QB's sound keywords only BEEP is implemented in FB. If PC speaker sound is required, it should be programmed using IN and OUT. See the example in the OUT keyword for a replacement of SOUND. There is a library called QBSound that allows to emulate qb's ability to PLAY in the background tunes encoded in strings, it uses the soundcard's synthesizer.

If what's required is to play .wav or .mp3 files thru a soundcard, external libraries as FMOD or BASS can be used in Linux and Windows. For DOS see the [DOS FAQ section](#).

[Back to top](#)

How do I access the serial ports?

DOS

See [DOS FAQ section](#).

Windows and Linux

See [Open Com](#).

[Back to top](#)

How do I print?

Since version 0.15 FB supports character output to printer.

To print graphics two approaches are possible:

- Preprocess the graphics data, program the printer, and send the data to it (see wikipedia.org/wiki/ESC/P). This is OS-portable but depends on the printer model. The only way for DOS, see also [DOS FAQ section](#).
- In Windows and Linux there are specific API calls. This is not OS portable but the OS's printer driver makes it printer-independent.

[Back to top](#)

How do I access the hardware ports?

INP, OUT and WAIT known from QB are implemented since version 0.1! of FB.

The GfxLib intercepts the calls to some VGA ports to emulate the widely used QB's palette manipulation and vsync methods. So ports &H3DA;, &H3C7;, &H3C8; and &H3C9; can't be accessed if GfxLib is used. All other ports are accessible.

No further tricks are required to use INP and OUT in Linux or DOS. For the Windows version the required device driver is installed each first time the program is run in a windows session; this requires Administrator rights for this first run or the program will end with an error. Note that accessing hardware ports by applications is not common practice in Windows and Linux.

[Back to top](#)

See also

- [Compiler FAQ](#)
 - [Frequently Asked FreeBASIC Graphics Library Questions](#)
- and
- [Win32 related FAQ](#)
 - [DOS related FAQ](#)

FreeBASIC on Xbox general questions

- Can FreeBASIC really make Xbox games?
- How was the FreeBASIC Xbox port made?
- How about a port for Xbox 360?
- How about a port for PlayStation or another console?
- Why don't you use an emulator until a developer gets a modded Xbox?
- Why don't you use the Microsoft XDK?
- Why don't you use the Microsoft debugger to fix it?
- Isn't this illegal? Can't Microsoft sue you?

Getting Started with FreeBASIC on Xbox questions

- What do I need to compile Xbox games with FreeBASIC?
- How would you get input?
- Does it only run on certain Xboxes?
- Is another language (eg C or ASM) needed for the job?
- Do you need a special lib?
- Can you use premade functions (inkey, line etc)?
- What else should I know?

FreeBASIC on Xbox general questions

Can FreeBASIC really make Xbox games?

In theory, yes. A copy of FreeBASIC 0.13 was ported to the Xbox in July 2005, and produced working executables. However, changes to the compiler for the 0.14 release broke compatibility.

The Xbox port is currently in zombie mode; nobody in the project team has the console at the moment - the original port was done by SJ Zero, but it got broken with the runtime library modifications done in v0.14.

The port is on hold until the GCC backend port is complete, because it is believed that this port will fix the Xbox port.

How was the FreeBASIC Xbox port made?

FreeBASIC for Xbox is possible because of the efforts of Open Source developers who created OpenXDK, the legal software development kit for Xbox. OpenXDK is created for a unixish environment, which is quite compatible with the FreeBASIC source.

The port was created by forcing the FreeBASIC runtime library to use the OpenXDK version of Glibc instead of the mingw32 version. When compiled with the correct flags, this creates what looks like a standard EXE file. CXBE then strips the Windows PE header on this executable file and replaces it with an Xbox header.

In effect, all the port really does is change the runtime library and link in a certain way to allow the CXBE utility to create an Xbox executable.

How about a port for Xbox 360?

The Xbox is an Intel Pentium 3 running a derivative of the NVIDIA nForce chipset, with an NVIDIA video chip and an NVIDIA SoundStorm sound card. This is why the Xbox port was possible and relatively straightforward to do.

The Xbox 360, on the other hand, uses an alien CPU, and similarly alien hardware. FreeBASIC cannot presently be made to produce executables for the Xbox 360.

Another problem is the lack of an equivalent to OpenXDK for the Xbox 360. This would force any port to use the Xbox 360 XDK, a copyrighted piece of software created by Microsoft. This would be illegal, immoral, and would put FreeBASIC in legal jeopardy.

Therefore, a port to the Xbox 360 is to be considered impossible at this time.

How about a port for PlayStation or another console?

The Xbox is an Intel Pentium 3 running a derivative of the NVIDIA nForce chipset, with an NVIDIA video chip and an NVIDIA SoundStorm sound card. This is why the Xbox port was possible and relatively straightforward to do.

The PlayStation, on the other hand, uses a RISC chip, which FreeBASIC cannot currently produce code for. Almost all consoles utilize non x86 processors, stopping development using FreeBASIC from being possible.

Another problem is the lack of an equivalent to OpenXDK for many consoles. This would force any port to use the commercial software development kit, a copyrighted piece of software created by the console manufacturer. This would be illegal, immoral, and would put FreeBASIC in legal jeopardy.

Therefore, a port to other consoles are to be considered impossible at this time. However, many ports to consoles and other platforms with legally available development kits will be possible when the GCC backend port is complete.

Why don't you use an emulator until a developer gets a modded Xbox?

No known Xbox emulator is capable of running FreeBASIC code. A legitimate hardware console is required to run the programs made. This makes an emulator completely useless for development.

Why don't you use the Microsoft XDK?

There are two main reasons not to use the Microsoft XDK.

1) Microsoft's XDK is a piece of copyrighted software, and utilizing it would be illegal and immoral, putting FreeBASIC at risk of legal action. Furthermore, no member of the FreeBASIC team has ever had any access to the Microsoft XDK, to prevent "tainting" FreeBASIC legally.

2) OpenXDK is developed around gcc and UNIX-style systems such as MinGW or Cygwin. This means that it can be integrated into FreeBASIC

with very little effort. Microsoft's XDK, on the other hand, is developed around Microsoft based compilers, and thus would not easily integrate in the source code of FreeBASIC.

NOTE: PROTECTION OF MICROSOFT'S COPYRIGHT, AND BY PROXY OF FREEBASIC, IS OF PRIMARY IMPORTANCE IN THIS PROJECT. WE DO NOT WANT HELP FROM ANYONE WITH THE XDK NOR DO WE WANT HELP FROM ANYONE WITH A DEBUGGER XBOX. ANY ATTEMPT TO OFFER THE XDK OR XDK RELATED HELP SHALL BE FORWARDED TO THE PROPER LAW ENFORCEMENT AGENCIES.

Why don't you use the Microsoft debugger to fix it?

There are two very good reasons not to use the Microsoft debugger.

1) Microsoft's XDK is a piece of copyrighted software, and utilizing it would be illegal and immoral, putting FreeBASIC at risk of legal action. Furthermore, no member of the FreeBASIC team has ever had any access to the Microsoft XDK, to prevent "tainting" FreeBASIC legally.

2) Microsoft's debugger requires a specially modified Xbox which neither SJ Zero nor any development team member has, and frankly, nobody who has worked on the port believes the debugger would work with FreeBASIC executables -- just as Microsoft's debugger can't read FreeBASIC debugger files, we doubt the Xbox debugger could read FreeBASIC debugger files. Regardless, point #1 trumps any attempt.

NOTE: PROTECTION OF MICROSOFT'S COPYRIGHT, AND BY PROXY OF FREEBASIC, IS OF PRIMARY IMPORTANCE IN THIS PROJECT. WE DO NOT WANT HELP FROM ANYONE WITH THE XDK NOR DO WE WANT HELP FROM ANYONE WITH A DEBUGGER XBOX. ANY ATTEMPT TO OFFER THE XDK OR XDK RELATED HELP SHALL BE FORWARDED TO THE PROPER LAW ENFORCEMENT AGENCIES.

Isn't this illegal? Can't Microsoft sue you?

Copyright is important for the protection of both commercial firms like Microsoft, and for small projects such as FreeBASIC. Without copyright, neither could enforce any rights over the code (In our case, such as the GPL). Generally speaking, it is copyright issues which are most often the cause of problems for open source projects attempting to do things like this.

Because the FreeBASIC Xbox port is created using software tools whose legality has already been established, themselves often derived from other sources whose legality has been established, FreeBASIC for Xbox is not illegal. Careful care has been taken to protect FreeBASIC from using any Microsoft copyrighted code, and diligence is and will be followed to prevent access to copyrighted code.

Getting Started with FreeBASIC on Xbox questions

What do I need to compile Xbox games with FreeBASIC?

The port isn't currently working, but when it is ready, you will only need a copy of FreeBASIC for Xbox.

How would you get input?

Initially, input will be acquired through SDL, as a gfxlib port is not yet complete. One of the developers is working on a generic SDL version of gfxlib, however, and it will provide full gfxlib functionality to the Xbox port.

Does it only run on certain Xboxes?

FreeBASIC for Xbox executables will only run on modded Xboxes. However, modding an Xbox is often as simple as loading a savegame in a certain game. More information is available on the [Xbox-Linux](#) website.

Is another language (eg C or ASM) needed for the job?

No. FreeBASIC for Xbox is the only thing needed.

Do you need a special lib?

No. FreeBASIC for Xbox will come with all supported libraries.

Can you use premade functions (inkey, line etc)?

Currently, input and output commands such as inkey and line aren't available, but all other functions, including file I/O, are. One of the developers is working on a generic SDL version of gfxlib, however, and if it functions, it will provide full gfxlib functionality to the xbox port.

What else should I know?

Executables created by FreeBASIC for Xbox are free of copyrighted Microsoft code, making them legal for distribution.

Windows and Linux source files which are designed to use SDL and rtlib will be capable of compiling for Xbox out of the box. While the Xbox does have keyboard support through the gamepad ports (proprietary USB connection), the input scheme will have to be altered to account for a gamepad.

DOS

The FreeBASIC port to DOS is based on the **DJGPP** port of the GNU to mode DOS.

The current maintainer of this port is **DrV**.

To be written: platform-specific information, differences from Win32/Linux tutorials, etc.

WANTED TESTERS

The DOS version/target of FreeBASIC needs more testers. If you are interested on DOS, please don't wait for future releases, give it a try now. Tests from old and new PC's are welcome (graphics, file I/O, serial port, ...). If something is broken, please file a detailed bug report into the forum or bug Tracker. If all works well, you are welcome to contribute as well. Make sure to test a recent version of FB (reports from FB older than 6 months are considered as obsolete and useless), and check this document **before** contributing.

Limitations

The DOS target is fairly well working and supported by FreeBASIC, and compared to other platforms exist, however. The features missing are mostly due to the operating system or DOS extender or C runtime:

- Cross-compiling to an other target
- Multithreading (see FAQ 23)
- Graphics in windowed mode or using OpenGL
- Setting **ScreenRes** to a size not matching any resolution supported by the operating system
- Unicode isn't supported in DOS, **wstring** will be the same as **zstring** and non-ASCII latin aren't supported. (do it yourself)
- Shared libraries (DLL's) can't be created/used (at least not "easily") and the number of external libraries usable with DOS is limited

FreeBASIC DOS related questions:

- 1. FB is a 32-bit compiler - do I need a 32-bit DOS?
- 2. What about FreeDOS-32? Does/will FB work, is/will there be a v
- 3. When running FreeBASIC in DOS, I get a 'Error: No DPMMI' mess
- 4. Is there a possibility how to get rid of this CWSDPMI.EXE and C
- 5. Can I use other DOS extenders, like DOS/4GW, Causeway, DOS
- 6. Where is the nice blue screen with all the ... / where is the IDE?
- 7. How can I view the documentation in CHM or PDF format in DO
- 8. How can I write/edit my source code?
- 9. How can I play sound in DOS?
- 10. How can I use USB in DOS?
- 11. How can I use graphics in DOS?
- 12. DEF SEG is missing in FB! How can I workaround this in my c
- 13. How can I rewrite QB's CALL INTERRUPT / access the DOS an
- 14. How can I rewrite QB's XMS/EMS handling?
- 15. FBC gives me a 'cannot find lsupcxx' error!
- 16. How can I use the serial or parallel port?
- 17. How can I use a printer?
- 18. How can I make a screenshot of a FreeBASIC program running
- 19. Graphics mode doesn't work (freeze / black screen / garbage c
- 20. Mouse trouble! Mouse doesn't work at all in DOS / arrow 'jump
- 21. What about the 64 KiB and 640 KiB problems / how much mem
- DOS?
- 22. My program crashes when I try to use more than cca 1 MiB RA
- FreeBASIC?
- 23. Threading functions are disallowed in DOS? Help!
- 24. Executables made with FB DOS are bloated!
- 25. Compilation is very slow with FB!
- 26. SLEEP doesn't work! How can I cause a delay?
- 27. The performance is very bad in DOS!
- 28. Can I access disk sectors with FB?
- 29. Can I use inline ASM with advanced instructions like SSE in D

See also

FreeBASIC DOS related questions

1. FB is a 32-bit compiler - do I need a 32-bit DOS?

No, the DOS version of FreeBASIC uses a **DOS extender**, allowing you of a 16 bit DOS kernel. You can use FreeDOS (16-bit), Enhanced-Dr-DC even MS-DOS down to version cca 4. You need at least 80386 CPU, se

2. What about FreeDOS-32? Does/will FB work, is/will there be a ve

FreeDOS-32 is experimental at time of writing, but it should execute Fre generated by it with no change. While FB DOS support already works or ready for FreeDOS-32 as well.

3. When running FreeBASIC in DOS, I get a 'Error: No DPML' messa

You need a DPML host (DPML kernel, DPML server), means the file "CW: HDPML32.EXE (cca 34 KiB). See requirements, and FAQ 4 for more det

4. Is there a possibility how to get rid of this CWSDPML.EXE and CV

Yes, 2 possibilities. To get rid of CWSDPML.EXE and create a standalon embedding CWSDPML, you need the CWSDPML package and the "EXE EXE2COFF, you remove the CWSDPML.EXE loader (file loses 2 KiB of : without extension), and then glue the file "CWSDSTUB.EXE" before this is cca 21 KiB bigger than the original one, but it is standalone, no additic rid of CWSDPML.SWP, you can then edit your executable with CWSPAR swapping (occasionally also - incorrectly - referred as paging). Note, how memory that can be allocated to the amount of physical memory that is i work can be done both with the FBC.EXE file and all executables create described in the CWSDPML docs in the package. Alternatively, you can r extender. They don't swap and create standalone executables. Since the Ring 0, the crash handling of them is not very good and can cause freez other hosts exit the "civil" way with a register dump. Also, spawning might WDOSX or D3X. Finally, you can use **HDPML** . Download the "HXRT.ZIP japheth.de/HX.html), extract "HDPML32.EXE" (cca 34 KiB) and "HDPML code, just for your information), and include it to your DOS startup ("HDF HDPML resident and prevent all FreeBASIC (also FreePASCAL and DJC crying about missing DPML and swapping. HDPML can **not** (easily / yet) executables. Running an executable containing D3X, CWSDPML or some HDPML or other external host is fine - the built-in host will be simply skip required for FreeBASIC, since it can't generate 16-bit real mode code, a to execute 32-bit code in DOS.

5. Can I use other DOS extenders, like DOS/4GW, Causeway, DOS/3

Not any extender around. So-called WATCOM-like extenders can't be used due to differences in memory management and executable structure. WDOSX are a multi-standard extenders, not only WATCOM-like. You also can use Tran's PMODE, nor PMODE/W (!), saves cca 5 KiB compared to CWSD EXE, but might affect stability or performance) or, as aforementioned, HI

6. Where is the nice blue screen with all the ... / where is the IDE?

The FreeBASIC project focuses on the compiler, generating the executable. It looks unspectacular, but is most important for the quality of software and does not include an IDE. There are several external IDEs for FreeBASIC, but none have a DOS version by now. If you really need one, you could try Rhide, but it is old and buggy, so use it at your own risk. See also FAQ 7 and 8.

7. How can I view the documentation in CHM or PDF format in DOS?

There is no good way to view CHM or PDF files in DOS by now. But you can view the documentation nevertheless. One of the FreeBASIC developers, coderJ, has a documentation viewer with the docs included in a special format, and it looks similar to the QB's built-in help viewer, but does not contain an editor. <http://www.execulink.com/~coder/FreeBASIC/docs.html>

8. How can I write/edit my source code?

There are many editors for DOS around, but only few of them are **good**. FreeDOS EDIT (use version 0.7d (!!)) or 0.9, 64 KiB limit, suboptimal stability (but works regularly), SETEDIT, INFOPAD (comes with CC386 compiler, can edit source code with highlighting for C and ASM, but not for BASIC).

9. How can I play sound in DOS?

There are 2 ways how to play sound in DOS: either the ("archaic") PC speaker, or a soundcard. The speaker is easy to control, but you can't think, even to play audio files (WAV, with decompression code also OGG). You can re-use most of existing QB code easily (example: o-bizz.de/qb...speaker.asm), but provides one channel and 6 bits only, and of course significant distortion. A soundcard, and, on some newest (P4) PC's the speaker quality is **very** better. For old ISA soundcards, there is much example code around, a new one is not easily accessed (supposing bare DOS in this category) either using a ("emulation" if it is available for your card (unfortunately, this is becoming more and more rare), or drivers are poor or even inexistent), or access the card directly (this is low-level hardware-related, assembler is also needed, and you need technical documentation). A few sources of inspiration like the DOS audio player MPXPLAY (written in

supporting both methods (native + "emu" drivers), see an up-to-date list drdos.org/...wiki...SoundCardChip. Support of sound in DOS is **not** by FB doesn't "support" sound on Win32 and Linux either - the games "con use FreeBASIC commands or libraries. To play compressed files (MP3, additionally need the decompressing code, existing DJGPP ports of those for this.

10. How can I use USB in DOS?

Again, not business of FB, you need a driver, FB doesn't "support" USB other Wiki: drdos.org/...wiki...USB about possibilities of USB usage in I

11. How can I use graphics in DOS?

GUI or graphics in DOS is definitely possible, there are several approaches

- Use the FB graphics library. It uses VESA (preferably linear) to access the video card and supports any resolution reported by driver, in addition to standard VGA modes.

Note: use preferably FB version **0.20** or newer, the FB DOS graphics work does **not work at all** in previous releases.

- VGA mode 320x200x8bpp: very simple, maximum reliability, resolution and 256 colours only, see example.
- VGA "ModeX" 320x240x8bpp: similar to above, less easy, compatibility, but low resolution and 256 colours only, see
- VGA "planed" mode 640x480x4bpp: difficult to set pixels, low compatibility, but low resolution and 16 colours only, no palette
- Some other "odd" VGA "ModeX" modes (like 360x240x8bpp only ;-)
- Write your own VESA code: More difficult, good compatibility possible, there might be reliability problems if not implemented
- Use an external library (DUGL, Allegro, MGL, WxWidgets) for graphics & GUI's, bloats EXE size, need to respect library reliability.

Note that some graphic cards report limited features through VESA, most example 8 MiB instead of 64 MiB) or less modes (for example only 24 bpp hidden, only lower resolutions visible (up to cca 1280x1024) while higher visible while "wide" modes hidden). This is a problem of the card, not of DOS, see the additional features in systems other than DOS, or in DOS only using going to the lowest level bypassing VESA.

12. DEF SEG is missing in FB! How can I workaroud this in my code
 DEF SEG is related to 16-bit RM addressing and was removed because accessing low memory areas is not possible, because FreeBASIC's runtime (DJGPP's) is not zero-based. For accessing low DOS memory, use DOS_SELECTOR, see "vga13h.bas" example, or "_dos_ds" selector for inline ASM, see example

```
' ' DOS only example of inline ASM accessing low memory
' ' Run in text mode 80x25 only

' ' Including dos/go32.bi will define "_dos_ds"
' ' "pointing" into G032 block

#include "dos/go32.bi"

Dim As UInteger DDS

DDS=_dos_ds

? : ? "Hello world !"
? "_dos_ds=$";Hex$(DDS)
? "This is just a tEst - abcd ABCD XYZ xyz @[`{ - pi

Do
  Sleep 1000
  If Inkey$<>"" Then Exit Do
  Asm
    mov  eax,[DDS] ' ' Directly using "_dos_ds" won't work
    push eax
    pop  gs        ' ' Just to get sure, it is usual
    Xor  ebx,ebx
    aa3:
    mov  al,[gs:0xB8000+2*ebx]
    cmp  al,65    ' ' "a"
    jb  aa1
    cmp  al,122   ' ' "z"
    ja  aa1
    cmp  al,90    ' ' "Z"
```

```

    jbe  aa2
    cmp  al,97  '' "a"
    jb   aa1
aa2:
    Xor  al,32  '' Swap case
aa1:
    mov  [gs:0xB8000+2*ebx],al
    inc  ebx
    cmp  ebx,2000
    jne  aa3
End Asm
Loop
? : ? "Bye"
End

```

13. How can I rewrite QB's CALL INTERRUPT / access the DOS and

Those interrupts can be accessed only using the DOS version/target of I

The access to interrupts is slower than in QB: with FB the DPMI host will switch, going to real-mode and coming back. All of that will eat hundre thousands of clocks if emm386 is loaded or if inside a Windows' DOS bc negligible or relevant, it depends. You should try to minimize the number more data per call - at least several KiB, not just one byte or a few bytes

Use DJGPP's DPMI wrapper:

```

#include "dos/dpmi.bi"

Type RegTypeX As __dpmi_regs

#define INTERRUPTX(v,r) __dpmi_int( v, @r )

```

Alternatively you can call INT's via inline ASM, 2 important things you ha that FB's memory model is not zero-based (see also FAQ 12, "DEF SEC

"direct" passing of addresses (like DS:[E]DX) to an INT will not work except with "DOS API translation".

14. How can I rewrite QB's XMS/EMS handling?

Depends why original code uses it. If it's just to bypass low memory limit "ordinary" FB's data types / memory handling features instead. If it is used out of luck and have to redesign the code completely, about sound see I preferably the low memory (should be no big problem, since the applications are in DPMI memory instead), DMA in DPMI memory is possible but more

15. FBC gives me a 'cannot find lsupcxx' error!

The source of this problem is the **libsupcxx.a** file in **LIB\DOS** directory, name. Your fault is to have extracted the ZIP with long file names enabled then using FB in DOS with no LFN support, resulting in this file looks **LIB** found. Rename the file in **LIBSUPCX.A** (one **X** only) or extract the ZIP again in FB 0.18, retest needed.

16. How can I use the serial or parallel port?

The DOS INT14 is not very useful/efficient as it sends/reads a single character use an external DOS32 comms library. /* does someone know a good one doesn't support OPEN COM on DOS target, coderJeff has an experiment included with FB since 0.18.3.

17. How can I use a printer?

DOS kernel won't help you here, so you have to prepare the text (trivial) easy for printers compatible with the "ESC/P" standard) yourself and send to parallel port or USB using an additional driver (see FAQ 10). So-called "can't be made working in DOS with reasonable effort.

18. How can I make a screenshot of a FreeBASIC program running

Ideally include this feature into your own code. DOS TSR based screenshots will work with text based screens, but probably none of them with FreeB really a bug on one or other side, it's a problem "by design".

19. Graphics mode doesn't work (freeze / black screen / garbage output)

Place a bug report into the forum. To make it as useful and productive as the following, proceed given steps and provide all related information:

- Check the limitations listed on the page GfxLib

- The graphics might not work well / at all on very old PC's. 500 MHz, provide exact info about it, if you don't know, use program to test.
- Exact info about your graphics card is needed. Test on DC (reports info only) and RayeR's VESATEST (also tries to inspect the result). Find out what "useful" modes (640 supported and with what bitdepths (8, 16, 24, 32 bpp), and look correctly.
- Find out and describe exactly what's wrong ("mode works FB", "no graphics but no error either", "black screen and fr messy/incomplete", ...).
- If some sophisticated program doesn't work, try also a mirror in middle of the screen.
- Try without a mouse driver (this reduces the CPU "cost").
- Find out what modes are affected. If a mode doesn't work, bitdepth, make sure to test the "cheapest"/safest modes 640x480 with 4 bpp, and 320x200 with 8bpp.
- For some old cards there are VESA drivers available (S3V and without, and include this info into your report.
- Remove potentially problematic content (memory management files. Nothing of such is required for FB, except a DPMI handler).
- Post info about your graphics card, CPU (if old), DOS type and a simple example code.

RayeR's VESATEST and CPUID can be downloaded here: rayer.ic.cz/ VBEDIAG here drv.nu/vbediag/.

20. Mouse trouble! Mouse doesn't work at all in DOS / arrow 'jumps

To use a mouse in DOS, you need a compatible driver, recognizing your FreeBASIC library. For optimal results, you need a **good** driver and a **su**

Mouse: the optimal choice, and pretty well available nowadays, is a PS/2 mouse. It can be a serial mouse, also this one should work. The newest is USB mouse, but its use in DOS, since it would need a compatible (INT33) high quality native driver, is not available by now, only some experimental), or rely on BIOS emulation (rather "unprecise").

Driver: the preferred choice is CTMOUSE from FreeDOS project. There

and 2.1b4 from 2008-July available. It is included with (but not limited to) version from here: ibiblio.org/pub/...mouse . None of them is perfect, but is better than most competitors. 1.9xx and 2.1xx will cooperate with BIOS 2.0xx bypasses BIOS and thus USB emulation will **NOT** work. Also Logitech a good job, download from here: uwe-sieber.de/util_e.html - version 6.0. Problems are DRMOUSE and some (old ?) versions of MSMOUSE.

If the mouse does not work at all, then most likely the driver is not loaded (see driver messages), or is not compatible with the INT33 "standard". Activating the "USB mouse emulation" in BIOS settings can help.

If the mouse control is "unprecise", the arrow "jumps" , then you either have a better one, or the BIOS emulation is bad - the solution is to buy a PS/2 mouse.

21. What about the 64 KiB and 640 KiB problems / how much memory does DOS?

Memory management is business of the DPMI host, rather than the DOS executables generated by it do **not** suffer from this problem, since they run in protected mode. You can use almost all the memory of your PC, with some exceptions above 64 or 640 KiB. CWSDPMI r5 is verified to work well up to 512 MiB (and does not crash it (unlike some older versions), but is silently ignored. HCWDPMI more: up to 4 GiB (the limit of 32-bit addressing), but there was not much memory on old machines - verified up to cca 1.5 GiB. FreeBASIC and code generated by it do not use DOS based memory managers (HIMEM/XMS and EMM386/EMS), but avoid them if they are present. All this of course applies to true DOS only, things like FreeDOS control over the memory management and provide only a small piece of memory (up to 64 MiB) to your DOS code.

22. My program crashes when I try to use more than cca 1 MiB RAM in FreeBASIC?

No, it's not a bug in FreeBASIC and it's not really DOS specific, see also the beginning, the easy solution is to use [shared](#) for arrays. More advanced use memory management functions like [Allocate](#). This is even more important for application to run on (old) PCs with little memory (and still edit at least small texts) as well as to use all huge RAM if available (and edit huge texts for example).

23. Threading functions are disallowed in DOS? Help!

The [Threading Support Functions](#) are not supported for DOS target, and will not be soon/ever. The reason is simple: neither the DOS kernel, nor the DPMI host

DOS Extender support threading, unlike the Win32 or Linux kernel. How DOS: you can set up your threading on the top of DPML. There are multi are:

- Set up an ISR, see "ISR_TIMER.BAS" example. This is not sufficient in some cases.
- There is a **pthread**s library for DJGPP allowing to "emulate" some degree. It works acceptably for [P]7-ZIP DJGPP port with FB yet.
- See forum [t=21274](#)

24. Executables made with FB DOS are bloated!

This is true but there is no easy/fast way to fix. FB is a 32-bit HLL compiler imported from DJGPP. !write me! (see forum: [t=11757](#))

25. Compilation is very slow with FB!

Problem: "FBC takes 10 seconds to compile a "Hello world" program ! T VBDOS / PowerBASIC do take < 1 second for the same job ..."

True, but this is "by design": FB compiles your sources in 3 steps, saving as described in CompilerCmdLine, while many older compilers do just 1 pass mostly to file I/O performance, see FAQ 27 below about possibilities of it. A small improvement can be achieved here by making the DPML host resident (**CWSDPML -p**, see FAQ 4 above). Note that the delay is mostly "additive" with bigger projects.

26. SLEEP doesn't work! How can I cause a delay?

sleep does work ... but has a resolution of cca 55ms = 1/18s only, thus "example using "SLEEP 2" for 2 milliseconds won't work. !write me! / !fix me!

- PIT / BIOS timer (runs at 18.2 Hz by default), peek the BIOS "ISR_TIMER.BAS" example, raise PIT frequency (use with **CWSDPML -p**)
- Poll the BIOS timer + PIT counter, method from TIMERHLL allows to enhance precision of above without raising the P
- RDTSC instruction (Pentium and newer)
- RTC clock
- Delay loops

27. The performance is very bad in DOS!

Problem: "The performance in DOS is poor compared to Win32 / Linux k same source !" or "Even worse, the very same DOS binary runs much fa !"

Both indeed can happen, nevertheless, DOS is no way predestined to be fixed. First you have to identify the area where you code loses perf

File I/O: DOS by default uses very little memory for its buffers, while oth and are "aggressive" with file caching. When dealing with many small file performance degrade. The solution is to install a filecache, for example l a RAMDISK (a good one: **SRDISK**) and copy the "offending" files (for e installation) there in and work there (make sure to backup your work to a regularly). Both will need an XMS host (use **HIMEMX**). Also DOS by de hard drives, while other systems try hard to find and use DMA. Test util: (Download: japheth.de/Download/IDECheck.zip) - run it in "I13" and "I results. If "DMA" is much faster (can be 1...10 times, depends from PC r DMA driver (for example **XDMA 3.1** is worth to try) can bring a big speed sure to read and write data in large pieces (16 KiB at least), not just sing more forgiving here, but on DOS every single file I/O call causes a small efficient code design with good buffering is crucial.

Graphics: Pentium 2 and newer CPU's have a cache related feature ca writes to video RAM. Drivers of other OSeS usually do enable it. DOS dc with graphics at all), neither does FB GFX. Use "VESAMTRR" tool by Ja "HXGUI.ZIP" package), it will enable the speedup, surviving also mode s application crashes, up to a reboot. The possible speedup factor varies i model, up to cca 20 times. Also the mouse handling eats some (too muc DOS, this is a known weak point (the design of DOS FB GFX is not "ver "standard" - which is not very good), fixing is theoretically possible but n several mouse drivers (see FAQ 20).

28. Can I access disk sectors with FB?

You can ... but FreeBASIC won't help you too much here: no "portable" s level way. For DOS 3 methods are possible

- Use logical disk access features of DOS for sector access see example in the forum: freebasic.net/forum/viewtopic
- Use physical disk BIOS INT 13, bypassing DOS
- Use CPU ports, lowest level, bypassing both DOS and BIC

freebasic.net/forum/viewtopic.php?t=16196, source of |
above, FASM forum or some OS development resources

Note that such experiments are a bit "dangerous" - you can easily lose c
unbootable if something goes wrong.

29. Can I use inline ASM with advanced instructions like SSE in DO

You can ... but SSE2 and above need to get enabled before. This is usu
the DPMI host, HDPMI32 and CWSDPMI 7 will do that, most other hosts
CUID for such instructions before using them. It's a good idea to provic
with older CPU's (early Pentium, 80386) besides supporting latest instru
those too.

See also

- [Compiler FAQ.](#)
- [FB Runtime Library FAQ.](#)
- [Frequently Asked FreeBASIC Graphics Library Questions](#)

Windows:

- Which IDEs are available for Windows?
- Can I get rid of the console / 'DOS' screen in a graphics application?
- My GUI program does nothing when run / The program compiles but I get a permission denied error in the linker
- How can I debug my program?
- Why Windows refuses to run my code using OUT and/or INP?
- I get the error 'Cannot start blah.exe because xxxx.dll was not found.' or similar. What is missing?
- Does FreeBASIC work with Windows Vista/7?
- Where can I find some tutorials on programming the Windows GUI?
- Are there Windows GUI code builders for FB?

FreeBASIC Windows questions

Which IDEs are available for Windows?

At the moment three full featured IDEs have been developed specifically for FB: **FBIde** (not being updated, avoid using of old versions of FBC bundled with it), **FbEdit** and **JellyFishPro**. These IDEs require a minimum configuration -as path to the compiler- to work.

You can also download **FBIde** and **FbEdit** as bundles (Editor + Compile that install in a single operation. But the bundled version of the compiler may be out of date.

Commercial "general use" IDEs can be used with FreeBASIC but may require an extensive setup. They are handy for multi language programming, as they provide a unified user interface.

Instructions for installing FB JFish Pro, FBIde, and FbEdit can be found

here:

[- IDE Installation guide for Windows](#)

[Back to top](#)

Can I get rid of the console / 'DOS' screen in a graphics application

Yes. You have to give FreeBASIC the right command for it when you compile your program.

- If you compile from a command prompt, simply add "-s gui" to the end, like "fbc myprg.bas -s gui"
- If you compile in a specific IDE, you have to edit the "Compiler Defaults".
 - In Jelly-Fish Pro, its "Compiler->Set Compiler Defaults->Compiler Options". Add "-s gui" (NO QUOTES) in that box.
 - In FbEdit select Windows GUI in the targets dropdown list in the right of the tool bar.

[Back to top](#)

My GUI program does nothing when run / The program compiles but I get a permission denied error in the linker

The problem may be related with the previous question. If a program tries to PRINT and it was compiled with "-s gui" it will freeze because no console is available. If the PRINT is issued before the first window is registered/opened, nothing will show in the screen or in the taskbar. The running program can only be seen in (and killed from) the task manager processes tab. If a new compilation is tried before killing the process it will give a "Permission denied" error when the compiler tries to modify a still running .exe.

In Windows GUI programs do not use console commands. Use MessageBox or print to a log file to issue any error message to the user. Be sure any PRINT to console you used for debugging is not compiled in the final version.

[Back to top](#)

How can I debug my program?

FreeBASIC can use any debugger compatible with GNU GDB. Insight Win32 debugger is an user friendly wrapper for GDB.

- Get Insight from [Dev-C++](#)
- Rename the file to Insight.tar.bz2, and decompress it to an empty folder
- Compile your program with the -g switch
- Run <Your_Insight_Dir>\bin\usr\bin\Insight.exe
- Do File>Open to load your program into Insight
- From there you can watch, set breakpoints, step, examine memory and registers. Check Insight's help

[Back to top](#)

Why Windows refuses to run my code using OUT and/or INP?

Windows requires a driver to be installed to access the hardware ports. FB-Win32 programs using INP and OUT include a built-in driver that installs temporarily for a session. Windows allows only users with Admin rights to run driver installations. This means if you usually run your windows sessions without Admin rights, you will have to use the window command line command RUNAS to run your program for the first time in each session so Windows allows it to install the driver.

If this behavior is not acceptable you can use an external library as [PortIO32](#) that installs a permanent port driver.

[Back to top](#)

I get the error 'Cannot start blah.exe because xxxx.dll was not found.' or similar. What is missing?

You are trying to run a program using a third party library that resides in dll not installed in your system.

FreeBASIC comes with headers and wrappers required to code for a lot of third party libraries but does not provide the actual runtime dll files.

You have to download and install these from their home page. Find in [the Links thread in the Libraries subforum](#) the URL's of the home pages of the libraries provided. You need the binaries for Win32 of the libraries

If you want to develop programs with the libs you will need the documentation too.

When releasing compiled code it is good etiquette to provide the third party dll's required to run it.

[Back to top](#)

Does FreeBASIC work with Windows Vista/7?

Yes. (Write me!!!)

[Back to top](#)

Where can I find some tutorials on programming the Windows GUI?

See the answers to this question in this [thread in the forum](#)

More advanced use requires a frequent consultation of the reference at the [Microsoft Developers Network](#). A local install of the API reference is possible, search Microsoft for the Platform SDK (a huge download) or install just the documentation.

[Back to top](#)

Are there Windows GUI code builders for FB?

Yes there are some 3rd party developments generating Windows API code from a windows designer à la Visual Basic:

[Jerry Fielden' Ezeegui](#) (freeware) uses a "graphical" textmode interface to let you build your code.

[mrhx Software's VISG](#) (GPL) has a more classical user interface.

Less helpful may be the graphical resource editors generating scripts for the resource compiler. Any editor generating scripts compatible with GoRC can be used, as the one included with [FbEdit](#). Graphical resource editors are a great help in designing dialogs and menus, but they leave to you the task of writing the window procedures required to make them active.

[Back to top](#)

See also

[Compiler FAQ](#)

[FB Runtime Library FAQ](#)

FreeBASIC Linux questions:

- FreeBASIC gives me an error 'ld: can't find -lX11' or something similar!
- How do I install FreeBASIC in Ubuntu?

FreeBASIC Linux questions

FreeBASIC gives me an error 'ld: can't find -lX11' or something similar!

FreeBASIC uses ld to link its files under linux. This program requires that any libraries you use have the '-dev' versions installed. For example, for the above error message, you'd want to install xlib-dev for your distribution. Other common errors are for glibc, which requires glibc-dev, and sdl, which requires sdl-dev. Most distributions make these easily available on your install media.

[Back to top](#)

How do I install FreeBASIC in Ubuntu?"

See [This thread in the FB forums](#)

[Back to top](#)

See also

[Compiler FAQ](#)

[FB Runtime Library FAQ](#)

Obsolete Keywords



Along the way FB has had a few keywords changed. Here is the list of those no longer supported. Old code must be updated if recompiled.

OPEN "CON:"

Use **Open Cons**

OPEN "ERR:"

Use **Open Err**

OPEN "PIPE:"

Use **Open Pipe**

POKEI

Use **Poke (Integer,Address,N)**

POKES

Use **Poke (Short,Address,N)**

SCREENINFO (Function returning a pointer to a structure)

Use **Screeninfo, Sub Returning Values In Its Arguments**

VAL64

Use **Vallng()**

GOSUB

Do not use **GoSub** in SUBs or FUNCTIONS anymore; allowed in -lang qb mode.

Brief definitions and explanations for words and phrases used in the FreeBASIC manual.

Index: [A](#) - [B](#) - [C](#) - [D](#) - [E](#) - [F](#) - [G](#) - [H](#) - [I](#) - [J](#) - [K](#) - [L](#) - [M](#) - [N](#) - [O](#) - [P](#) - [Q](#) - [R](#) - [S](#) - [T](#) - [U](#) - [V](#) - [W](#) - [X](#) - [Y](#) - [Z](#)

A

access rights

The level of access associated with **type** or **class** members. Public members are accessible to any code; protected members are accessible to member functions and any derived **type** or **class** member functions; private members are accessible only to member functions of that **type** or **class**. By default, **type** members have public access rights, while **class** members are private.

any pointer

A variable or expression that points to a memory address where it is not known, at least from the compiler's point of view, what type of data is stored at that address. In C this would be the same as a void pointer or (void *). See **Ptr**.

archive

An archive is a group of files or a single file packed into a container format and usually compressed before or afterward. Typical container formats are GNU Tar and Zip. Typical compression formats are Gzip and Zip.

argument

Data that is passed to a procedure. The procedure refers to this data using the parameter(s) in its parameter list.

argument passing convention

The method in which arguments are passed to procedures, being either **By Reference** or **By Value**. See **Passing Arguments to Procedures**.

array (container)

A collection of data whose elements are stored contiguously in memory (one after the other, in increasing order). Because of this, an array offers random-access to its elements (any element can be accessed at any time). Insertion or removal of elements anywhere but at the back of the container requires that those elements that follow be relocated, so a linked-list is typically preferred when insertion or removal needs to be efficient.

assembler

A component in the tool chain for translating source code in to executable programs. The assembler converts the low level assembly instruction mnemonics emitted by the compiler to object code.

assignment

Assignment is one of the fundamental operations of computing. All it means is copying a value into the memory location pointed at by a variable. The value might be a literal, another variable, or the result of some expression. For an instance of a **Type** or **Class**, this involves calling one of its assignment operators. Not to be confused with initialization.

automatic storage

Refers to storage on the call stack. Local procedure variables, objects and arrays with automatic storage are allocated when the procedure is called, initialized when defined, destroyed (in the case of objects) when leaving the scope they're declared in and deallocated when returning from the procedure.

automatic variable/object/array

A variable, object or array with **automatic storage**.

[Back to top](#)

B**byref**

ByRef specifies passing arguments to procedures by reference. Arguments passed by reference can be modified by the procedure and the changes seen by the caller.

byval

ByVal specifies passing arguments to procedures by value. Procedures receive a copy of the argument passed. With **Type** or **Class** instances, this involves instantiating temporary objects by calling their copy constructor. These temporaries are destroyed upon procedure exit.

binaries

Binaries are the end result of source code. Binaries include executable files (.exe on windows), static library files (.a), dynamic library files (.dll on windows, .so on Linux), and relocatable object files. (.o)

.BSS section

The part of the executable program that will contain zero bytes only when the program starts. Since all of the bytes are zero, the final size of the executable can often be reduced by placing uninitialized data, or zero initialized data in this section.

buffer

A region of memory that allows data to be saved or manipulated before being copied somewhere else. In a communications device this may hold incoming or outgoing data yet to be processed. In graphics, a buffer may contain an image before being copied to the screen.

[Back to top](#)

C

call back

A control mechanism where a caller lets a procedure call another procedure (the call back) provided by the caller typically through a function pointer.

call stack

A chunk of memory reserved for a process or thread that is used as a stack for storing various information needed by procedures when they are called. Among the information stored on the call stack are all of the local automatic variables, objects and array data and usually whatever parameters are passed to the procedure. These items are allocated (*pushed* onto the call stack) when the procedure is called and deallocated (*popped* from the call stack) when the procedure returns, either by the caller or the callee, depending on the calling convention used. The initial and maximum sizes of this reserved memory vary by platform.

caller

A misnomer used to refer to the point in code in which a procedure is called.

cast

A cast operation changes one data type to another using specified rules. A **Type** structure can implement a custom **cast** for any intrinsic data type and/or other TYPES, See **Cast**.

code block

Several lines of source code grouped together all sharing at least one common scope. For example a procedure's code block will be all the lines of code between **Sub** and **End Sub**.

com port

A short name for serial communications port. A program can communicate with an external device, such as modem or another computer through a com port (nowadays the good old com ports are deprecated in favor of USB). See **Open Com**.

compiler

A compiler is a computer program which takes source code and transforms it into machine or object code.

compiler directives

These are instructions included in the text of the program that affect the way the compiler behaves. For instance the compiler might be directed to include one section of code or another of depending on the target

operating system.

compound statement

A statement composed one or more additional statements. Typically, a compound statement has a beginning (opening statement), a middle (a statement block) and an end (closing or ending statement), while some have additional parts. Examples of compound statements would be **If** and **Function**.

constant

A symbol that retains a consistent value throughout the execution of the program. See **Const**.

constructor (module)

A special type of module-level procedure that is automatically called prior to the module-level code flow. See **Constructor (Module)**.

constructor (TYPE or CLASS)

A special member function of a **Type** or **Class** that is called when an object is instantiated.

CVS

Concurrent Versions System. The file manager implemented at Sourceforge where sources are stored, it keeps the history of the changes introduced by the developers. Used by FB in the past. (see also SVN and GIT)

[Back to top](#)

D

.DATA section

The part of the executable program that will data that can be changed while to program is running.

debugger

A program that allows controlled execution of compiled code. The values of variables can be tracked, execution can be paused, stepped or accelerated, etc. A debugger is typically used to help find the source of programmer errors in source code, called 'bugs'.

declaration

A source code statement that introduces a symbol, constant, variable, procedure, data type, or similar, to the compiler but not necessarily allocate any space for it. See [Dim](#), [Declare](#), [Extern](#), [Type](#).

definition

A source code statement (or statements) that allocates space for data or code. For example, [Sub](#) defines a procedure by allocating space for the program code it will contain. Some statements can be both a declaration and a definition. For example, [Dim](#) both declares and defines a variable.

dereference

The act of obtaining a value from memory at a given address. See [Operator * \(Valueof\)](#), [Pointers](#).

descriptor

Refers to the internal data structure used by the compiler and runtime library for managing variable length strings and arrays.

destroy (TYPE or CLASS)

The act of deconstructing and deallocating memory for an object instance. When an object is destroyed, its destructor is called. This happens automatically when an object goes out of scope, or when [Delete](#) is called with a pointer to an object.

destructor (module)

A special type of module-level procedure that is automatically called at program termination. See [Destructor \(Module\)](#).

destructor (TYPE or CLASS)

A special member function of a [Type](#) or [Class](#) that is called when an object is destroyed.

dll

Shorthand for **dynamically linked library**.

DPMI

A method / standard allowing to execute protected mode code (mostly also 32-bit) on a 16-bit real mode DOS kernel. Affects only DOS version of FreeBASIC. See also [DOS related FAQ](#)

DJGPP

A complete 32-bit C/C++ development system for Intel 80386 (and higher) PCs running DOS and includes ports of many GNU development utilities.

dynamically linked library

A file containing executable code that is loaded by another application when it is started. Also referred to as a **dll** or shared library. See [Shared Libraries \(DLLs\)](#).

[Back to top](#)

E

enum

A data type restricted to a sequence of named values given in a particular order. See [Enum](#).

executable

A binary file that can be run. It consists of libraries and object files bound together by the linker.

exit sub/function

When called inside a procedure, leaves the procedure and returns control to the calling program.

expression

An instruction to execute a statement that will evaluate/return a value.

[Back to top](#)

F

field

Commonly refers to a data member in a **Type** or **Class**.

file number

An integer associated with an open file or device as given in **open**. All subsequent operations on the opened file or device must use the same file number.

format string

A sequence of characters that controls how data should be presented. See **Format**, **Print Using**.

function

A procedure defined using **Function**, optionally taking parameters and returning a value.

function pointer

A variable containing the address of a function. The address (function) to which the variable points can be changed while the program is running allowing for dynamic program flow, such as call back functions.

[Back to top](#)

G

get/put buffer

See: Image Buffer. An image buffer in FreeBASIC's native format.

GIT

The file manager implemented at Sourceforge where sources are stored it keeps the history of the changes introduced by the developers. Used by FB now. (see also CVS , SVN and Git).

global variable

A variable that is visible to all procedures within a module, across multiple modules, or both. See [Common](#) and [Extern](#).

GNU

A mass collaboration project with the primary goal to provide a free and non-proprietary Unix-like operating system.

GPL

Short hand for GNU General Public License: a license for software and other kinds of works. Open source, obligates the user to keep the project open source and under the GPL.

graphics primitive

A graphics primitive is another term for common shapes like circles and rectangles.

[Back to top](#)

H**hash table**

A data structure that associates keys with values allowing for efficient look-up of values based on a given key.

header

When talking about a collection of data, this is generally the first part of that data that describes the rest. When talking about (header) files, this refers to an include file. In FreeBASIC the file extension '.bi' is usually used.

heap

The area of memory (free store) provided by the runtime library (and operating system) from which the program can dynamically allocate memory. See [Allocate](#).

[Back to top](#)

I

image buffer

A collection of data used to describe an image, containing such information as width, height, color depth and pixel data.

include file

A kind of source file that typically contains type definitions and declarations for variables and procedures that one or more other source files refer to. In general, these files provide a public interface to some module or modules, although a file that is **#included** can contain anything whatsoever.

initialization

The act of giving a variable a value at the point of its creation. For object instances, this involves calling one of its constructors. Not to be confused with assignment, which gives an already existing variable another value.

instance

An instantiated object of a **Type** or **Class**.

instantiate

The act of creating an object of a **Type** or **Class**, either directly with **Dim**, **new**, or **newinstance**, or indirectly by, for example, passing an object to a procedure by value.

[Back to top](#)

J

[Back to top](#)

K

[Back to top](#)

L

library

Compiled code stored in a single file that can be used when making other programs. A library typically has one or more headers (or include files) to provide all the needed declarations for using the library.

linked list (container)

A collection of data whose elements are typically stored on the heap. The linked list's elements store the addresses of their adjacent elements, and so only sequential access (an element is accessed by following the links from adjacent elements) is possible. This scheme does provide constant time insertion of elements anywhere into the container, however, and because of this is often preferred over the array.

linker

A program which combines multiple modules and libraries into a single executable which can be loaded into the computer's memory and followed by the computer. FreeBASIC uses the **LD** linker. Linkers are the most common, but not the only way to produce executables.

LGPL

Shorthand for GNU Lesser General Public License. Like the GNU GPL, but more permissive allowing non-(L)GPL'd works to be statically linked to the LGPL'd work, provided that the new work can have the LGPL'd portion relinked or replaced.

local variable

A variable that is visible only within the scope in which it is declared, and that is destroyed when program execution leaves that scope.

lock

A synchronization mechanism such that only one thread or process can have access to a shared object, for example a global variable, a device,

or a file.

[Back to top](#)

M

member

A data field, procedure, enumeration, type alias or anything else declare within a **Type** or **class** definition.

member data

Variables associated with a **Type** or **class**. Member data can be static or non-static.

member function

A procedure associated with a **Type** or **class**. Member functions have full access rights to the members of its type or class, and can be static or non-static.

method

See **member function**.

module

A source file in its entirety, including any `include` files that may be present as well. Typically, a module is a logical unit of code, containing parts of a program that relate to one another. For example, if making a game, one may separate the procedures needed for error logging from the procedures that control graphics into their own modules.

[Back to top](#)

N

non-static member data

Member data that each instance of a **Type** or **class** gets their own copy

of.

non-static member function

A **member function** that has an implicit **This** reference as an argument.

null

A constant usually associated with pointers denoting a 'nothing' value. This value is typically an integer '0' (zero) - the 'NULL terminator' appended to zstrings is `chr(0)`, or `asc(!"0")` - but can also be defined as pointer type, like `cast(any ptr, 0)`.

[Back to top](#)

O

object code

Code in machine-readable form that can be executed by your computer's CPU and operating system, usually linked with libraries to create an executable file.

operand

One of the arguments passed to an operator. For example, in the expression `a = b + c`, the operands are `a`, `b` and `c`, while the operators are `=` and `+`.

operator

A function taking one or more operands (arguments) and returning a value. Operators can work on built-in data types, or can be overloaded to work on user defined types. See [Operators](#).

overload

To declare a procedure having the same name as another, but with different parameters. Free functions, or module-level functions, can be overloaded using the **Overload** keyword. **Type** or **Class** member functions can be overloaded by default.

[Back to top](#)

P

page buffer

A buffer used for holding the contents of the screen before being displayed on screen. Where multiple page buffers are allowed, one page will be visible to the users while all others are hidden. Also the active page (the one to which changes are made) need not be the visible one allowing changes to one page while showing another.

parameter

The name used by a procedure that corresponds to the argument that is passed to it.

parameter list

The parenthesized comma-separated list of parameters in a procedure declaration or definition.

PDS

Professional Development System. Sometimes referred to as QB7.1.

pitch

The number of bytes per row, in an image or screen buffer. If there is no padding between rows, then this can be calculated by `width * bytes_per_pixel`, but this is not necessarily safe to assume. The screen's pitch can be found using [ScreenInfo](#), and an image buffer's pitch can be found by checking the `pitch` value in the image's header.

pointer

A data type used to hold addresses. The kind of pointer determines how the data at the address is interpreted when the pointer is dereferenced, or when used with [Operator -> \(Pointer To Member Access\)](#). See [Pointers](#).

preprocessor

The FreeBASIC preprocessor is responsible for expanding Macros and replacing Defined values with their values.

procedure

A generic name for any block of code that can be called from somewhere else in a program. See [Sub](#), [Function](#).

property

A property is a special sort of type/class members, intermediate between a field (or data member) and a method. See [Property](#).

ptr

Shorthand for pointer. See [pointer](#).

[Back to top](#)

Q**queue (container)**

A collection of data that offers first-in first-out (FIFO) storage and retrieval. Typically, elements can only be inserted at the back and removed from the front but can be accessed from either end.

[Back to top](#)

R**ragged array (container)**

A ragged array is an array having rows of differing lengths.

real number

Any positive or negative number including fractions, irrational and transcendental numbers (like π or e) and zero. Variables containing a real number have a limited range and precision depending on the number of bits used to represent the number. See: [Single](#) and [Double](#).

registers

Places inside the CPU for data storage. 80386 and compatible 32-bit models have EAX, EBX, ECX, EDX, ESI, EDI, EBP and ESP, plus some special (control/test/debug) registers. **NOT** related to "Windows registry"

[Back to top](#)

S

scope

Refers to the life-time and visibility of some component of the program, like a variable or a procedure. For example, a variable defined inside a procedure would have procedure scope: it is visible throughout the procedure, but not outside the procedure's code block. When the procedure ends, the variable goes out of scope and no longer exists.

scope block

A code block where all the lines of source have the same scope. An explicit scope block can be indicated with the **Scope** statement. Scope blocks may also be implicit with the usage of **If . . Then**, **For . . Next**, and other compound statements.

shared library

A library that exists once on a system that multiple executables can link to at runtime. See [Shared Libraries \(DLLs\)](#).

source code

Code written by the programmer, in a human-readable form, not yet compiled.

stack (container)

A collection of data that offers last-in first-out (LIFO) storage and retrieval. Typically, elements can only be inserted, accessed and removed from the top of the stack.

statement block

One or more lines of code bookended by a compound statement.

static library

A library that is linked into a program at link time. There is one copy of the library for each executable that links to it. All data is executable specific. See [Static Libraries](#).

static member data

Member data that each instance of a [Type](#) or [Class](#) shares. This data is defined outside of any [Type](#) or [Class](#), and takes up no space in the resulting object instance.

static member function

A **member function** without an implicit `this` reference as an argument. Static member functions can be called normally through a variable, or directly using the type's name and the scope resolution operator See [Static \(Member\)](#).

static storage

Refers to storage in the `.BSS` or `.DATA` sections of an executable. Variables, objects and arrays with static storage are allocated and initialized at compile-time and destroyed (in the case of objects) and deallocated at program-termination. Explicitly initialized variables, objects and arrays are allocated in the `.DATA` section.

static variable/object/array

A variable, object or array with **static storage**.

sub

A procedure defined using [Sub](#), optionally taking parameters and not returning a value.

SVN

Subversion. A version control system that allows users to keep track of changes made to sources and documents. Used by FB in the past. (see also CVS and GIT)

SWIG

A tool that automatically translates C headers to FreeBASIC (although not always perfectly).

symbol

Used to refer to variables, labels, functions, methods, procedures, or other programmatic constructs in a program.

[Back to top](#)

I

.TEXT section

The part of the executable program that will contain program instructions and constant data.

this reference

A reference to an instance of a **Type** or **Class** that is passed as a hidden argument to non-static member functions of that type or class.

Throughout the member function, this instance is referred to using the `this` keyword, See [This](#).

thread

A thread of execution within a process (running program) that shares execution time with other threads in the same process. See [Threading](#).

trace

To follow the execution of a program step-by-step either manually by examining the source code, or more practically with a debugger.

[Back to top](#)

U

union

A structure that can be used to store different types of variables, such as integers, doubles and fixed-length strings in the same location, but only one at a time. See [Union](#).

user defined data type

A **Type**, **Union**, **Enum**, or **Class** data type.

[Back to top](#)

V

variable

A symbol representing data in memory.

VBDOS

Visual BASIC for DOS, a historical BASIC compiler by M\$ from 1992, following after QBASIC. DOS platform dropped very soon, VBDOS never became popular.

vector

A series of data items in memory that can be accessed by an index number. Similar to an array except that vector elements are not necessarily all contained within a single block of memory.

[Back to top](#)

W

warning

A message displayed by the compiler during compilation that suggests there may be potential problems with the current code.

wiki

An on-line system that provides a set of pages containing information that can be viewed and modified by the public. In this context, it is typically used to refer to the FreeBASIC on line documentation.

[Back to top](#)

X

x86

Refers to the instruction set compatible with the 8086 (and later) CPU architecture, FreeBASIC only supports 80386 and later.

[Back to top](#)

Y

[Back to top](#)

Z

zstring

A zstring is in essence a standard C style string terminated by a null character. This data type is provided for greater compatibility with C libraries.

[Back to top](#)

Data

- Data
- Read
- Restore

Debugging

- Assert
- AssertWarn
- Stop

Hardware Access

- Inp
- LPrint
- Lpos
- Out
- Wait

Operating System

- Beep
- Sleep
- End (Statement)

Stub Pages

- As
- For
- To
- Is
- Step

Control Flow

- Do
- End If
- If
- Loop
- Next
- Then
- Until
- Wend
- While

Uncategorized

- End (Block)
- OffsetOf
- SizeOf
- TypeOf
- Let
- Rem
- Option()

Runtime Error Codes



Runtime error codes and messages used by the runtime library.

Description

Freebasic returns the following runtime error codes:

0	No error
1	Illegal function call
2	File not found signal
3	File I/O error
4	Out of memory
5	Illegal resume
6	Out of bounds array access
7	Null Pointer Access
8	No privileges
9	interrupted signal
10	illegal instruction signal
11	floating point error signal
12	segmentation violation signal
13	Termination request signal
14	abnormal termination signal
15	quit request signal
16	return without gosub
17	end of file

No user error code range is defined. If **Error** is used to set an error code it is wise to use high values to avoid collisions with the list of built-in error codes. (This built-in list may be expanded later.)

See also

- **Err**
- **Error**

- On Error
- **Error Handling**

Comparison of C/C++ and FreeBASIC



C/C++

FreeBASIC

variable declaration

```
int a;  
int a, b, c;
```

```
dim a as integer  
dim as integer a, b, c
```

uninitialized variable

```
int a;
```

```
dim a as integer = any
```

zero-initialized variable

```
int a = 0;
```

```
dim a as integer
```

initialized variable

```
int a = 123;
```

```
dim a as integer = 123
```

array

```
int a[4];  
a[0] = 1;
```

```
dim a(0 to 3) as integer
```

```
a(0) = 1
```

pointer

```
int a;  
int *p;  
p = &a;  
*p = 123;
```

```
dim a as integer
```

```
dim p as integer ptr
```

```
p = @a
```

*p = 123

structure, user-defined type

```
struct UDT {  
  int myfield;  
}
```

```
type UDT  
myfield as integer  
end type
```

typedef, type alias

```
typedef int myint;
```

```
type myint as integer
```

struct pointer

```
struct UDT x;  
struct UDT *p;  
p = &x;  
p->myfield = 123;
```

```
dim x as UDT  
dim p as UDT ptr  
p = @x  
p->myfield = 123
```

function declaration

```
int foo( void );
```

```
declare function foo( ) as integer
```

function body

```
int foo( void ) {  
  return 123;  
}
```

```
function foo( ) as integer  
return 123  
end function
```

sub declaration

```
void foo( void );
```

```
declare sub foo( )
```

sub body

```
void foo( void ) {  
}
```

```
sub foo( )  
end sub
```

byval parameters

```
void foo( int param );  
foo( a );
```

```
declare sub foo( byval param as integer )  
foo( a );
```

byref parameters

```
void foo( int *param );  
foo( &a );  
  
void foo( int& param );  
foo( a );
```

```
declare sub foo( byref param as integer )  
foo( a )
```

statement separator

```
;
```

```
:
```

end-of-line

for loop

```
for (int i = 0; i < 10; i++) {  
...  
}
```

```
for i as integer = 0 to 9
```

```
...
```

```
next
```

while loop

```
while (condition) {  
...  
}
```

```
while condition
```

```
...
```

```
wend
```

do-while loop

```
do {  
  ...  
} while (condition);
```

```
do
```

```
...
```

```
loop while condition
```

if block

```
if (condition) {  
  ...  
} else if (condition) {  
  ...  
} else {  
  ...  
}
```

```
if condition then
```

```
...
```

```
elseif condition then
```

```
...
```

```
else
```

```
...
```

```
end if
```

switch, select

```
switch (a) {  
  case 1:  
    ...  
    break;  
  case 2:  
  case 3:  
    ...  
    break;  
  default:  
    ...  
    break;  
}
```

```
select case a
```

```
case 1
```

```
...
```

```
case 2, 3
```

```
...
```

```
case else
```

```
...
```

```
end select
```

string literals, zstrings

```
char *s = "Hello!";  
char s[] = "Hello!";
```

```
dim s as zstring ptr = @"Hello!"
```

```
dim s as zstring * 6+1 = "Hello!"
```

hello world

```
#include <stdio.h>  
int main() {  
    printf("Hello!\n");  
    return 0;  
}
```

```
print "Hello!"
```

comments

```
// foo  
/* foo */
```

```
' foo  
/' foo '/
```

compile-time checks

```
#if a  
#elif b  
#else  
#endif
```

```
#if a  
#elseif b  
#else  
#endif
```

compile-time target system checks

```
#ifdef _WIN32
```

```
#ifdef __FB_WIN32__
```

module/header file names

```
foo.c, foo.h
```

```
foo.bas, foo.bi
```

typical compiler command to create an executable

```
gcc foo.c -o foo
```

```
fbcc foo.bas
```

Comparison of integer data types: FreeBASIC vs. C/C++ (using GCC)



	C int	C long long [int]	C long [int]	FB Long	FB LongInt	FB Integer
32bit win32	32	64	32 (ILP32)	32	64	32
32bit linux-x86	32	64	32 (ILP32)	32	64	32
64bit win64	32	64	32 (LLP64)	32	64	64
64bit linux-x86_64	32	64	64 (LP64)	32	64	64

See also

- [Creating FB bindings for C libraries](#) - How to translate C data types to FB

This area of the Wiki is for documenting everything about the compiler and the runtime libraries. It is, however, incomplete. If you find that information provided here does not match what the source is doing then please update the relevant pages here. New pages and articles may be added freely, provided they help understanding what's going on inside FB.

Developing FreeBASIC Itself

Compiling a Development Version of FreeBASIC

Getting the source code

Compiling FB for DOS

Compiling FB on Linux

Compiling FB on Windows

Getting source code updates and recompiling FB

Debugging FB

FB build configuration options

Known problems when compiling FB

GCC toolchain choice

Running the FreeBASIC test suite

Normal vs. Standalone

Glossary

Notes on the creation of FB releases

FB and cross-compiling

Bootstrapping/cross-compiling fbc

Creating FB bindings for C libraries

C Header Translation Tutorial

Header Style Guidelines

External Libraries Index (header status)

Compiler internals

Quick overview of all modules

The objinfo feature
Memory management
Lexer & preprocessor
Parser & compiler (fb, parser, symb, rtl)
Purpose
Top level parsing process
Symbols
Representation of data types
SELECT CASE
Profiling FB programs
Structure packing/field alignment

Run-time (rtlib) and Graphics (gfxlib2) Libraries

Keyboard input: inkey(), multikey(), etc.
Overview of drivers (backends)
Pixel formats

Compiler

The FreeBASIC compiler (fbc) is released under the **GPL** license.

Libraries

With the exception of **LibFFI** (which is used for **Threadcall**), the runtime is released under the **LGPL** license, for both the single-threaded and multi-threaded versions, with this extension to allow linking to it statically:

"As a special exception, the copyright holders of this library give you permission to link this library with independent modules to produce an executable, regardless of the license terms of these independent modules, and to copy and distribute the executable under terms of your choice, provided that you also meet, for each independent module, the terms and conditions of the license of that module. If you link this library with other modules, and you modify this library, you may extend this exception to your version of the library, but you are not obligated to do so. If you do not wish to do so, delete this exception statement from your version."

The Gfx library (libfbgfx) is released under the **LGPL** license.

LibFFI is released under the following license, found at <http://github.com/atgreen/libffi/blob/master/LICENSE>:

```
libffi - Copyright (c) 1996-2011 Anthony Green, Red Hat, Inc and  
See source files for details.
```

```
Permission is hereby granted, free of charge, to any person obtaining  
a copy of this software and associated documentation files (the  
``Software''), to deal in the Software without restriction, including  
without limitation the rights to use, copy, modify, merge, publish,  
distribute, sublicense, and/or sell copies of the Software, and to  
permit persons to whom the Software is furnished to do so, subject to  
the following conditions:
```

```
The above copyright notice and this permission notice shall be  
included in all copies or substantial portions of the Software.
```

```
THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,  
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF  
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT
```

IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Documentation

The documentation is released under the [GFDL](#) license.

About the FreeBASIC project.

The FreeBASIC project is a set of cross-platform development tools initially created by Andre Victor, consisting of a compiler, GNU-based assembler, linker and archiver, and supporting runtime libraries, including a software-based graphics library. The compiler, *fbcc*, currently supports building for i386-based architectures on the DOS, Linux, Windows and Xbox platforms. The project also contains thin bindings (header files) to some popular 3rd party libraries such as the **C runtime library**, **Allegro**, **SDL**, **OpenGL**, **GTK+**, **the Windows API** and many others, as well as example programs for many of these libraries.

FreeBASIC is a high-level programming language supporting procedural, object-oriented and meta-programming paradigms, with a syntax compatible to **Microsoft QuickBASIC**. In fact, the FreeBASIC project originally began as an attempt to create a code-compatible, free alternative to QuickBASIC, but it has since grown into a powerful development tool. FreeBASIC can be seen to extend the capabilities of QuickBASIC in a number of ways, supporting more data types, language constructs, programming styles, and modern platforms and APIs.

BASIC compatibility

- FreeBASIC is not a "new" BASIC language. You don't need to learn much new if you are familiar with any Microsoft-BASIC variant. You can use either "-lang qb" for compatibility, or (default "-lang fb" for some of the new features, but it also brings some restrictions and some similarity with the "C" programming language. See also CompilerDialects.
- FreeBASIC is case-insensitive; explicit "main" procedure is not required; most of the graphic and console statements and procedures found in Microsoft QuickBASIC are implemented, et cetera.
- Only with "-lang qb": scalar variables don't need to be dimensioned and suffixes can be used; line numbers are supported; **GoSub** supported.

Clean syntax

- Only a small number of keywords have been added. All procedures are implemented as libraries, so for the most part, there are no new intrinsic routines, and therefore there is a low chance of having name duplication with old code.

Thin bindings (header files) to existing C libraries and APIs

- No wrappers or helpers are necessary, just a ported header file, making usage of external C libraries very easy
- The official distribution comes with several bindings to existing C libraries already, see **External Libraries TOC** for a complete up-to-date list

Multi-platform

- FreeBASIC currently runs on 32-bit Windows, Linux, and DOS (a 16-bit DOS is good enough, although FreeBASIC itself and compiler output are 32-bit) and also creates applications for the Xbox console. More platforms to come.
- The runtime library was written with portability in mind. All third-

party tools used exist on most operating systems already as they are from the GNU binutils. The compiler is written in 100% FreeBASIC code (that is, FreeBASIC compiles itself.), which makes it simple to be bootstrapped as it doesn't depend on non-portable tools.

Unicode support

- Besides ASCII files with **Unicode** escape sequences (\u), FreeBASIC can parse UTF-8, UTF-16LE, UTF-16BE, UTF-32LE and UTF-32BE source (.bas) or header (.bi) files, they can be freely mixed with other sources/headers in the same project (also with other ASCII files).
- Literal strings can be typed in the original non-Latin alphabet, just use a text-editor that supports some of the **Unicode** formats listed above.
- The **wString** type holds wide-characters, all string procedures (like **Left**, **Trim**, etc) will work with wide-strings too.
- **Open** was extended to support UTF-8, UTF-16LE and UTF-32LE files with the **Encoding** specifier. **Input #** and **Line Input #**, as well as **Print #** and **Write #** can be used normally, and any conversion between **Unicode** to ASCII is done automatically if necessary.
- **Print** also supports **Unicode** output (see Requirements).

A large number of built-in data types

- Integer: **Byte**, **UByte**, **Short**, **UShort**, **Integer**, **UInteger**, **Long**, **ULong**, **LongInt**, **ULongInt**
- Floating-point: **Single**, **Double**
- String: fixed, variable-length or null-terminated (**zString**), up to 2GB long
- **Unicode** strings (**wString**), like **zString**, but with support for wide characters. Use the Windows **Unicode** API procedures directly, etc.

User-defined types (UDTs)

- Unlimited nesting.
- BASIC's **Type** statement is supported, along with the new **Union**

- statement (including anonymous nested unions).
- Array fields utilizing up to eight dimensions can be used.
- Procedure pointer fields.
- Bit fields.

Enumerations (enums)

- Easily declare a list of constants with sequential values with **Enum**.

Arrays

- Fixed- and variable- length arrays are supported, up to 2 GB in size.
- Up to eight dimensions, including arrays with unknown dimensions.
- Any lower and upper boundaries.
- Element data can be preserved during a re-size of variable-length arrays with **ReDim** using the new **Preserve** specifier.

Pointers

- Pointers to any of the data types listed above, including string characters, array elements and UDT's.
- Uses the same syntax as C.
- Unlimited indirection levels (e.g., pointer to pointer to ...).
- Procedure pointers.
- Indexing []'s (including string indexing).
- Type casting.

Variable, object and array initialization

- For static, module-level or local variables, arrays and UDT's.

Default procedure parameter values

- For numeric, string and UDT parameter types.

Procedure overloading

- Including procedures with default parameter values.

In-line assembly

- Intel syntax.
- Reference variables directly by name; no "trick code" needed.

Traditional preprocessor support

- Same syntax as in C.
- Single-line macros supported with the `#define` command, including parameters.
- Multi-line macros supported with the `#macro` command.

Type aliases

- Supporting forward referencing as in C, including UDT and procedure pointer types.

C-like escape sequences for string literals

- Same as in C (except numbers are interpreted as decimal, not octal).

Debugging support

- Full debugging support with *GDB* (the GNU debugger) or *Insight* (a *GDB* GUI frontend).
- Array bounds checking (only enabled by the `-exx` command-line option).
- Null pointer checking (same as above).

Create OBJ's, LIB's, DLL's, and console or GUI EXE's

- You are in no way locked to an IDE or editor of any kind.
- You can create static and dynamic/shared libraries adding just on command-line option (`-lib` or `-dylib/-dll`).

As a 32-bit application

- FreeBASIC can compile source code files up to 2 GB in size.
- The number of symbols (variables, constants, et cetera) is only limited by the total memory available during compile time. (You can, for example, include OpenGL, GTK/SDL, BASS, simultaneously in your source code.)

Optimized code generation

- While FreeBASIC is not an optimizing compiler, it does many kinds of general optimizations to generate the fastest possible code on x86 CPU's, not losing to other BASIC alternatives, including the commercial ones.

Completely free

- All third-party tools are also free. No piece of abandoned or copyrighted software is used (except GoRC on Win32). The assembler, linker, librarian/archiver, and other command-line applications come from the GNU binutils programming tools.

- **Standard Data Types**
- **User Defined Types**

Different ways angles are measured



Written by RandyKeeling

This very simple tutorial assumes that you know what an angle is.

There are three commonly used ways to measure the size of an angle:

- Degrees (deg)
- Radians (rad)
- Gradients (grad)

Degrees

Most people are familiar with angles measured in degrees. A full circle has 360 degrees. There are two different ways, degrees decimal and DMS (degree, minute, second)

We can always show a degree as we would any decimal number by showing the decimal part. For example, 75.23° means that we have 75 degrees and twenty-three hundredths of a degree.

In the DMS system, each degree is made up of 60 minutes (or arcminutes) and 60 seconds (or arcseconds) and is marked with a double quote. So a degree measured as 75 degrees, 14 minutes, 52 seconds.

To convert DMS to decimal degrees you can use the following code.

```
Dim D As Integer
Dim M As Integer
Dim S As Integer
Dim DD As Single

'' Convert to degree decimal
DD = D + M / 60 + S / 3600 '' 3600 comes from 1/60
```

Radians

Radians are more common in computer programming and mathematics. The constant Pi (often given the symbol of the lowercase Greek letter pi). Pi notation never ends) and is the circumference of any circle divided by the diameter (places) is $\text{Pi} = 3.1415926535897932385$. The value of Pi can also be found using the formula:

$$\text{Pi} = 4 * \text{Atn} (1)$$

With the radian system, a full circle has $2 * \text{Pi}$ (6.2831853071795864770). Pi is often given the symbol of the lowercase Greek letter pi. Pi notation never ends) and is the circumference of any circle divided by the diameter (places) is $\text{Pi} = 3.1415926535897932385$. The value of Pi can also be found using the formula:

To convert between radians and degrees (decimal) you can use the following formulas:

```
Const PI As Double = 3.1415926535897932

Dim D As Double
Dim R As Double

R = D * PI / 180      '' A full circle has 360 degrees
D = R * 180 / PI
```

The value of Pi is used so often, it is not uncommon to find it defined in a program as a useful constant.

```
Const PI As Double = 3.1415926535897932
Const TWO_PI As Double = 6.283185307179586
Const HALF_PI As Double = 1.570796326794896
Const DegToRAD As Double = 0.01745329251994330
Const RADToDeg As Double = 57.29577951308233
```

Gradients

Gradients are used mainly in some forms of engineering. Within the gra

A Brief Introduction To Trigonometry



Written by RandyKeeling

This tutorial includes:

- Right Triangles
- Pythagoras' Theorem
- Trigonometric Functions
- Applying Trigonometric functions
- Inverse Trigonometric functions
- Other Trigonometric functions
- Law of Sines, Law of Cosines, and other relationships

Trigonometry can be thought of as the study of triangles. There is more to it than that, but this will suffice for this tutorial. While this may seem to be of limited use, many problems in both the real and virtual worlds can be solved by creative application of triangles.

A triangle has three sides and in 'normal' (i.e. Euclidean) space has three angles whose measurements add to be exactly 180 degrees (or Pi radians). For this tutorial we will deal only with 'normal' triangles (for those interested in other spaces, search for non-Euclidean triangles or non-Euclidean geometry).

Right Triangles

To begin with, we will deal with a special class of triangles known as right triangles. A right triangle has one angle that measures 90 degrees. Because the angles of a triangle must be exactly 180 degrees, there can be only one 90 degree angle in a triangle (and it is the largest angle in a right triangle). Below is FreeBASIC code to draw an image of a right triangle. (This image will be referred to throughout the tutorial.) In this image, uppercase letters denote sides, and their corresponding lowercase letters denote the angle opposite of the side. For example, angle *y* is the angle opposite side *Y*.

ScreenRes 640,480,8

```
'Triangle  
Color 7  
Line (220,140) - (220,340)  
Line (220,140) - (420,340)  
Line (220,340) - (420,340)
```

```
'right angle  
Color 12  
Line (220,320) - (240,320)  
Line (240,320) - (240,340)
```

```
'angles  
Color 13  
Locate 20,29  
Print "x"  
Locate 42,50  
Print "y"
```

```
'Sides  
Color 14  
Locate 31,43  
Print "Z"  
Locate 31, 26  
Print "Y"  
Locate 45, 40  
Print "X"
```

```
Sleep
```

The box in the lower right hand corner means that it is a right angle (measures 90 degrees). The side opposite of that angle (side Z) is called the hypotenuse and is the longest side in a right triangle.

Pythagoras' Theorem

Perhaps the first bit of trigonometry that most people learn is the relationship commonly known as Pythagoras' Theorem. It simply states that the square of the hypotenuse of a right triangle is equal to the sum of the square of the other two sides. It is easier to understand in equation form.

$$Z^2 = X^2 + Y^2$$

A trivial example application of this law might be the following.

If player one is 100 meters due east of a marked location (the origin) and player two is 150 meters due north of the same location, how far apart are they?

$$D = \text{SQR}(100^2 + 150^2)$$

Trigonometric Functions

Long ago people discovered that regardless of the size of the triangle, certain ratios were always the same. For example, in the image of the triangle above, if the measure of angle y is 45 degrees, then regardless of the size of the triangle, the ratio Y/X will always be the same. Collections of these ratios are trigonometric functions.

The three primary functions are Sine (**Sin**), Cosine (**Cos**), and Tangent (**TAN**). There are many different ways to define these three functions. One way is with relationships between sides of a right triangle.

- Sine (**Sin**) is the ratio of the side opposite the angle in question to the hypotenuse. In the above triangle, the sine of the angle y (written as $\text{SIN}(y)$) is the length of side Y divided by the length of side Z .
- Cosine (**Cos**) is the ratio of the side adjacent to the angle in question to the hypotenuse. In the above triangle, the cosine of angle y (written $\text{COS}(y)$) is the length of Side X divided by the length of side Z .
- Tangent (**Tan**) is the ratio of the side opposite to the angle in question to the side adjacent to the angle in question. In the above

triangle, the tangent of angle y (written as $\text{Tan}(y)$) is the length of side Y divided by the length of side X .

Many people remember these relationships with the mnemonic device SOHCAHTOA (pronounced Sow Cah Toe-a) which is of course Sin = opposite/hypotenuse, Cos = adjacent/hypotenuse, and Tan = opposite/adjacent.

FreeBASIC has functions for these trigonometric functions and others.

Applying Trigonometric functions

Referring again to the triangle image above, let's say that player one is on the ground at the point near angle y and player two is at the point near angle x (off of the ground). If player one knows how far he or she is from the side Y (let's say 25.2 meters) and can measure the value of angle y (let's say 31.5 degrees) how far off the ground is player two? How far away is player one from player two?

To solve this we look at what pieces of information we know. We know the adjacent side to angle y (25.2 meters) and the measure of angle y (31.5 degrees). This is enough information to use the tangent function. $\text{Tan}(y) = \text{Opposite}/\text{adjacent}$, or $\text{TAN}(31.5 \text{ degrees}) = \text{Opposite}/25.2 \text{ meters}$. Using a little algebra to rearrange this we get $\text{opposite} = \text{Tan}(31.5 \text{ degrees}) * 25.2 \text{ meters}$. To find the distance between the players we could use Pythagoras's Theorem now that we know the two non-hypotenuse sides of the triangle or we could use the cosine. Using cosine would give $\text{Cos}(y) = \text{adjacent}/\text{hypotenuse}$. With some algebra we get, $\text{hypotenuse} = 25.2/\text{Cos}(31.5 \text{ degrees})$.

Before we can write a program to solve this, we must remember that FreeBASIC, like most programming languages, works with radians, not degrees (see [Angles](#)).

In FreeBASIC we could get the answer with this code.

```
Const PI As Double = 3.1415926535897932
Dim Opposite As Double
```

```
Dim Hypotenuse As Double
Dim Angle As Double

Angle = 31.5 * Pi / 180

Opposite = Tan ( Angle ) * 25.2
Hypotenuse = 25.2 / Cos ( Angle )

Print Opposite
Print Hypotenuse

Sleep
```

The above code tells us that player two is about 15.4 meters off the ground and around 29.5 meters away (along the hypotenuse).

Inverse Trigonometric functions

But what if you know the sides of a triangle and need to find the angle? You would then use the inverse trigonometric functions.

- ArcSine (or Inverse Sine)
- ArcCosine (or Inverse Cosine)
- ArcTangent (or Inverse Tangent)

For example, using the above set-up, if player two was 30 meters off the ground and 50 meters away from player one (along the hypotenuse) what is the measure of angle y ? Looking at our trigonometric functions it looks like we have need of the sine function (an opposite and a hypotenuse).

Sin (y) = opposite/hypotenuse, ArcSine (opposite/hypotenuse) = y .

```
Print Asin (30/50)
```

This gives an angle of about 0.6435 radians, or around 36.9 degrees.
The FreeBASIC command for each of these inverse functions are:

- **Asin** (arcsine)
- **Acos** (arccosine)
- **Atn** (arctan, there is also **Atan2** which takes the opposite and adjacent sides of the triangle, not their ratio)

Other Trigonometric functions

There are other trigonometric functions that are defined in terms of the above functions. Although none of the below are defined in FreeBASIC.

- Secant (sec(y)) is $1/\mathbf{Cos}(y)$
- Cosecant (csc(y)) is $1/\mathbf{Sin}(y)$
- Cotangent (cot(y)) is $1/\mathbf{Tan}(y)$

Each of these has an inverse (or arc) functions as well.

Law of Sines, Law of Cosines, and other relationships

All of the above has assumed a right triangle, but this was an aid in explaining the basic trigonometric functions. The following does not rely on right triangles; these identities are valid for any triangle.

Law of Sines

$$\mathbf{Sin}(y)/Y = \mathbf{Sin}(x)/X = \mathbf{Sin}(z)/Z$$

Law of Cosines

$$Z^2 = X^2 + Y^2 - 2*X*Y*\mathbf{Cos}(z)$$

Other Identities

$$\mathbf{Sin}^2(y) + \mathbf{Cos}^2(y) = 1$$

This means the same as $\mathbf{Sin}(y)*\mathbf{Sin}(y) + \mathbf{Cos}(y)*\mathbf{Cos}(y) = 1$

$$\mathbf{Tan}(y) = \mathbf{Sin}(y)/\mathbf{Cos}(y)$$

There are several more useful identities out there. Search for trigonometric identities or consult any higher mathematical reference.

x86 Microprocessor Architecture



x86 or 80x86 is the generic name of a microprocessor architecture first developed and manufactured by Intel.

More information can be obtained by reading [this](#) Wikipedia article.

Structure packing/field alignment



The default layout of `Type` and `Union` structures in FreeBASIC is compatible to that of GCC, following the SysV (Linux/BSD) and Microsoft (Windows) ABIs. This allows for binary compatibility with GCC and other compilers.

By default, fields are aligned to their natural boundaries, which are:

- A multiple of 1 for 1-byte data types
- A multiple of 2 for 2-byte data types
- A multiple of 4 for 4-byte data types
- A multiple of 4 for 8-byte data types (32bit x86 DOS(DJGPP)/Linux/BSD)
- A multiple of 8 for 8-byte data types (Win32/Win64, 32bit ARM Linux, 64bit x86_64/AArch64 Linux/BSD)
- The largest natural boundary of the fields of `Type/Union` data type:
- Dynamic string descriptors are handled as `Type` structures with the data pointer field being the one with the largest natural alignment
- Fixed-length strings are aligned according to the alignment required for the character size.
- Static arrays are aligned according to the alignment required for the element data type.

The compiler aligns fields by inserting padding bytes in front of them in order to move them to an offset that corresponds to their natural boundary, or to a multiple of the value given with `Field = N`, if it is smaller than the field's natural alignment. On the x86 architecture, such proper alignment is not required but can result in better performance when accessing the fields. Other architectures might actually require proper alignment.

In addition to field alignment, the whole structure's size is rounded up to multiple of the largest natural alignment of its fields, by adding padding bytes at the end of the structure. This ensures that in an array of such structures, each individual one is properly aligned as required by the fields.

Using the GFX_NULL driver in Windows

The client area of the window is updated using GfxLib. Menus, toolbars

```
' ' Example of use of the GFX_NULL driver in windows
' ' The GfxLib is set up in the ON_Create sub
' ' The GFXLib buffer is drawn to screen in th On_Pa
' ' The GfxLib is updated in the event loop
```

```
#include "fbgfx.bi"
#include once "windows.bi"
```

Using fb

```
Dim Shared bmi As bitmapv4header
Dim Shared mywin As rect
```

```
' '
' '-----
```

```
Function on_paint(ByVal hwnd As HWND,ByVal wparam As
```

```
    Dim rct As RECT
    Dim pnt As PAINTSTRUCT
    Dim hDC As HDC
```

```
    'draw the gfx buffer to screen
    hDC = BeginPaint(hwnd, @pnt)
    GetClientRect( hwnd, @rct )
    With rct
        StretchDIBits hDC, 0, 0, .Right-.Left+1, .bottom
        .bottom-.top+1,ScreenPtr,CPtr(bitmapinfo
    End With
```

```
    EndPaint hwnd, @pnt
```

```

    Function = 0

End Function

''
''-----
Function on_Create(ByVal hwnd As HWND,ByVal wparam As WPARAM,ByVal lparam As LPARAM) As Boolean
    Dim rct As RECT
    'set a gfxscreen of the size of the client area
    GetClientRect( hwnd, @mywin)
    ScreenRes mywin.right+1,mywin.bottom+1, 32, 1, (mywin.right+1)*mywin.bottom+1
    'and create a bmp header,required to paint it yourself
    With bmi
        .bv4Size = Len(BITMAPV4HEADER)
        .bv4width=mywin.right+1
        .bv4height=-(mywin.bottom+1) 'negative value
        '(standard BMP's are bottom to top)
        .bv4planes= 1
        .bv4bitcount=32
        .bv4v4compression=0
        .bv4sizeimage=mywin.right+1*mywin.bottom+1*4
        .bv4RedMask = &h0F00
        .bv4GreenMask = &h00F0
        .bv4BlueMask = &h000F
        .bv4AlphaMask = &hF000
    End With

    Function = 0

End Function

''
''-----
Function on_Destroy(ByVal hwnd As HWND,ByVal wparam As WPARAM,ByVal lparam As LPARAM) As Boolean
    'clear arrays....
    PostQuitMessage( 0 )

    Function = 0

```

```
End Function
```

```
''  
''-----
```

```
Function WndProc ( ByVal hWnd As HWND,ByVal message  
                  ByVal wParam As WPARAM,ByVal lParam
```

```
    Function = 0
```

```
    Select Case As Const message
```

```
    Case WM_CREATE
```

```
        Function = On_create(hWnd,wparam,lparam)
```

```
    Case WM_PAINT
```

```
        Function = On_paint(hWnd,wparam,lparam)
```

```
    Case WM_DESTROY
```

```
        Function = On_destroy(hWnd,wparam,lparam)
```

```
    Case Else
```

```
        Function = DefWindowProc( hWnd, message, wParam
```

```
    End Select
```

```
End Function
```

```
''  
''-----
```

```
''main program create window + event loop
```

```
Dim wMsg As MSG
```

```
Dim wcls As WNDCLASS
```

```
Dim szAppName As ZString * 30 => "Random Rectang
```

```
Dim hWnd As HWND
```

```
Dim i As Integer
```

```
With wcls
```

```
    .style = CS_HREDRAW Or CS_VREDRAW
```

```
    .lpfnWndProc = @WndProc
```

```
    .cbClsExtra = 0
```

```
    .cbWndExtra = 0
```

```
    .hInstance = GetModuleHandle( null )
```

```
    .hIcon = LoadIcon( NULL, IDI_APPLICATION
```

```

        .hCursor          = LoadCursor( NULL, IDC_ARROW)
        .hbrBackground   = GetStockObject(WHITE_BRUSH)
        .lpszMenuName     = NULL
        .lpszClassName   = @szAppName
    End With

    If( RegisterClass( @wcls ) = FALSE ) Then
        End
    End If

    'make a non-resizable screen
    hWnd = CreateWindowEx( 0, szAppName, "Example of C
        WS_OVERLAPPEDWINDOW And Not (WS_sizebox Or v
        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, (
        NULL, NULL, wcls.hinstance, NULL )

    ShowWindow( hWnd, SW_NORMAL )
    UpdateWindow( hWnd )

    While 1
        If PeekMessage( @wMsg, NULL, 0, 0, PM_Remove)
            If wmsg.message=WM_QUIT Then
                Exit While
            End If
            TranslateMessage( @wMsg )
            DispatchMessage( @wMsg )
        Else
            'update the gfx buffer
            Line (Rnd*mywin.right, Rnd*mywin.bottom)
            RGB(Rnd*255, Rnd*255, Rnd*255), bf
            redrawwindow (hwnd, 0, 0, rdw_invalidate)
        End If
    Wend

    End wMsg.wparam

```

A worldwide standard for storing, categorizing and interpreting characters

Unicode is an industry standard designed to allow text and symbols from all of the writing systems of the world to be consistently represented and manipulated by computers. Developed in tandem with the Universal Character Set standard and published in book form as The Unicode Standard, Unicode consists of a character repertoire, an encoding methodology and set of standard character encodings, a set of code charts for visual reference, an enumeration of character properties such as upper and lower case, a set of reference data computer files, and rules for normalization, decomposition, collation and rendering.

The Unicode Consortium, the non-profit organization that coordinates Unicode's development, has the ambitious goal of eventually replacing existing character encoding schemes with Unicode and its standard Unicode Transformation Format (UTF) schemes, as many of the existing schemes are limited in size and scope, and are incompatible with multilingual environments. Unicode's success at unifying character sets has led to its widespread and predominant use in the internationalization and localization of computer software. The standard has been implemented in many recent technologies, including XML, the Java programming language, and modern operating systems.

Common Unicode formats include:

- **UTF-8**
- **UTF-16**
- **UTF-32**

GET/PUT image header example



Example showing the two different headers used for image buffers.

Note: `ImageInfo` is provided as a simpler alternative to reading the image

```
' fbgfx.bi contains the necessary structures and co
' directly with image headers
#include "fbgfx.bi"

' in lang fb, structures and constants are containe
#if __FB_LANG__ = "fb"
Using FB
#endif

' function to show info on an image
Sub show_image_info( ByVal image As Any Ptr )
    Dim As PUT_HEADER Ptr header
    Dim As Integer w, h, bpp, pitch

    header = image
    If( header->Type = PUT_HEADER_NEW ) Then

        Print "New style header"

        w = header->Width
        h = header->height
        bpp = header->bpp
        pitch = header->pitch

    Else

        Print "Old style header"

        w = header->old.width
        h = header->old.height
        bpp = header->old.bpp
    End If
End Sub
```

```

        pitch = w * bpp

    End If

    Print "Image dimensions are " & w & "*" & h
    Print "Image uses " & bpp & " bytes for each pixel"
    Print "A row of image pixels takes " & pitch & " bytes"

End Sub

Dim As Any Ptr picture

ScreenRes 320, 200, 32

picture = ImageCreate( 10, 10, RGB(128, 192, 255) )

Put( 40, 40 ), picture, PSet

show_image_info( picture )

ImageDestroy picture

Sleep

```

NOTE: To use this code with an array, pass your array to the function, like

```
show_image_info( VarPtr( myarray( L ) ) )
```

where L is the lower bound of myarray().

Getting Started



This is a good introduction to FB for QBasic programmers, based on SJ

Getting started with the software

You can download FreeBASIC here: <http://www.freebasic.net/index.php/>
And FBIDE here: <http://fbide.sourceforge.net/>

When installing FBIDE, select "FBIDE only," to not install the old version package.

When running FBIDE the first time, you will have to browse to find the FI computer.

Hello World!

Open up FBIDE and type the following:

```
PRINT "Hello World!"  
SLEEP
```

Now press F5. Congratulations, you've just seen how much like QB Free can use most console commands for QB just like you remember. For ex

```
LOCATE 10,10  
PRINT "I'm the center of the universe!"  
SLEEP
```

The Amazing Screen 13

Now, put "SCREEN 13" before your code, to see how easy it is to use g

```
SCREEN 13  
PRINT "Hello World!"  
SLEEP
```

From there, all of the standard QB graphics commands work as you remember. Here is an example:

```
SCREEN 13
LINE (1,1)-(100,100),1,bf
PRINT "Hello World!"
CIRCLE (10,10),10,2
PSET (30,15),3

SLEEP
```

FreeBASIC also has new graphics features. For example, QB has never had a fullscreen mode. Try running this program:

```
SCREEN 15
LINE (1,1)-(100,100),1,bf
PRINT "Hello World!"
CIRCLE (10,10),10,2
PSET (30,15),3

SLEEP
```

After opening a graphics window via the SCREEN command, you can change between windowed and fullscreen modes.

Another nice feature of the graphics library in FreeBASIC is that you can use video mode. The following code demonstrates this.

```
DIM as integer page
DIM as integer notpage
DIM as integer a, b

screen 12, , 2 'This sets the screen for 2 pages
notpage = 1 'This sets the backpage

DO
```

```

IF page = 0 THEN page = 1 ELSE page = 0 'These two lines flip
IF notpage = 1 THEN notpage = 0 ELSE notpage = 1 ' backpage

SCREENSET page, notpage 'This flips the page

CLS 'First we clear the screen
b = b + 1
IF b > 100 THEN b = 0
FOR a = 1 TO 128
  PSET (b,a),a 'Then we draw a line. It moves without flickering
NEXT a

LOOP UNTIL INKEY = CHR(27)

```

This works for any mode, so you can use the high resolution modes for flipping, using standard QB graphics commands!

Why ASM is No Longer Required

I wouldn't be saying this if it wasn't true. Using ASM in BASIC to increase program speed is no longer necessary. Ignoring SDL, Allegro, DirectX, OpenGL you've got the above page flipping and advanced graphics modes at your disposal, Inkey, which we've all grown to love or hate, but there are also two new things QBers have had to resort to assembly code to do since the dawn of time.

```

DIM as integer x, y, buttons
CONST as integer escapeKey = 1
SCREEN 12

WHILE NOT MULTIKEY(escapeKey) 'this checks the escape key every
  GETMOUSE x, y, , buttons 'This gets the mouse state
  PRINT x,y,buttons
WEND

```

With this knowledge, you should be able to begin programming in FreeBASIC that it entails; Speed, power, and portability!

Jason K. Firth, SJ Zero
<http://www.qbxi.net>

Who I Am

I am a hobbyist software developer whose tool of choice is FreeBASIC. I have Quest for a King, Nietzsche, Star Phalanx, and Rambo vs. Kitty Cat under my belt.

What I Do

I'm an instrumentation engineering technologist, not a programmer.

Contact Me

Don't.

This is an excerpt from an article published in QBXL Magazine, with per

FreeBASIC's greatest strength is it's ability to seamlessly integrate with C, while maintaining the ease of use that is QB. Even before FB had a built-in graphics and sound routines, we were using SDL to get graphics and sound routines working. Before the release of Winsock, a number of coders, myself included, fought with the head of FreeBASIC. Today, I'm just going to cover how to get started with three : tinyPTC. After understanding the fundamentals, you'll see that using C li few exceptions, C libraries are no more difficult to use in FreeBASIC tha

What are these Libraries, Anyway?

These libraries are particularly useful because they tend to provide funct

SDL is a library with graphics and input support built in, and a bunch of s **font support**, and **audio**. It can be used with OpenGL, but I won't be co

TinyPTC is primarily a graphics library, the simplest one available. It doe to the graphics reigon to draw to.

FMod is a 3d sound and music library. Though its license is strange, it w and it nicely encapsulates 3D sound.

Including the Library

The first step in getting any of these libraries to work is including their he For SDL, it's simply

```
'$INCLUDE: "SDL\SDL.bi"
```

For FMOD, it's

```
'$Include: 'fmod.bi'
```

and for tinyPTC, you'll want

```
'$INCLUDE: 'tinyptc.bi'
```

'2. Initializing the library, loading a file'

Obviously, you can't just include the lib and fire away if it's got to do stuff
To initialize SDL and load a bitmap into memory, you must:

```
CONST SCR_WIDTH = 640
CONST SCR_HEIGHT = 480
DIM MenuScreen AS SDL_Surface ptr 'our bitmap
DIM Shared video AS SDL_Surface ptr 'our screen surface

SDL_Init ( SDL_INIT_VIDEO )
video = SDL_SetVideoMode( SCR_WIDTH, SCR_HEIGHT, 32, 0 ) 'sets tl
MenuScreen = SDL_LoadBMP("bitmap.bmp")
```

To initialize FMOD and load a sound into memory, you must:

```
DIM sound AS INTEGER 'it's just a handle, so it's an int!

IF FSOUND_GetVersion <= FMOD_VERSION THEN
ErrorQuit "FMOD version " + STR$(FMOD_VERSION) + " or greater req
End If

If FSOUND_Init(44100, 32, 0) = FALSE Then
ErrorQuit "Can't initialize FMOD"
End If

sound = FSOUND_Sample_Load(FSOUND_FREE, "sound.wav", FSOUND_HW3D,
```

Finally, there's no data formats to load with tinyPTC because it's so simple

```
const SCR_WIDTH = 320
const SCR_HEIGHT = 200
const SCR_SIZE = SCR_WIDTH*SCR_HEIGHT

if( ptc_open( "tinyPTC test", SCR_WIDTH, SCR_HEIGHT ) = 0 ) then
end -1
end if
```

Blitting, Playing, or Plotting

The most important step, obviously, is to get whatever you want to do to relatively easy, and can be encapsulated further into a wrapper function. screen means going:

```
SUB BlitImage(x as integer,y as integer,image as sdl_surface ptr,
DIM Rectangle as SDL_Rect
DIM Rectangle2 as SDL_Rect

Rectangle.X = 0
Rectangle.Y = 0
rectangle.w = image->w
rectangle.h = image->h
Rectangle2.x = x
Rectangle2.y = y

SDL_Blitsurface image, @rectangle, dest, @rectangle2

END SUB
```

For FMOD, the steps to play a sound aren't that difficult either:

```
FUNCTION fModPlayWave( samp1 as integer ) AS INTEGER
'where samp1 is the number returned by FSOUND_SampleLoad

DIM position(0 to 2)' as FSound_Vector
DIM vel(0 to 2)' FSound_Vector

fModPlayWave = FSOUND_PlaySoundEx(FSOUND_FREE, samp1, NULL, TRUE

END FUNCTION
```

And TinyPTC, which is again, not a high level library like the other two, c code:

```
SUB putd(BYREF buffer(), BYVAL x AS INTEGER, BYVAL y AS INTEGER,
buffer((y * SCR_WIDTH) + x) = colr
ptc_update @buffer(0) 'This is a pageFlip

END SUB
```

Shutting Down

So you don't have to manage memory and do all the boring mundane tasks down the library before your program exits. Luckily, all three programs automatically shut it down, the library no longer cares. It's beautiful.

```
SDL: SDL_Quit ()
```

```
fmod: FSOUND_Close ()
```

```
tinyPTC: PTC_Close ()
```

That's all there is to quitting!

As you can see, there is nothing inherently more difficult in using libraries in QuickBASIC. In fact, because coders don't need to jump through hoops, it's easier, even with the more modern OS and hardware.

After doing some searches, I quickly noticed that there simply wasn't an example to the FreeBasic Community. As you know A Windows Console is a windows console, which means it's created with the use of the Windows API accessed from the Console Window. So There's no need to turn the mouse on or Set the X and Y coordinates and get the states of the mouse buttons. This is a tutorial.

- Getting Mouse Coordinates:

The mouse cursor, when the mouse is moved, continuously updates its position where the pointer currently is on the screen.

- Setting Mouse Coordinates:

For some reason there may be a need to position the mouse pointer at a specific location on the screen.

- Getting The Mouse Button Statuses:

Quite simply, when the user presses a button on the mouse, it returns a value indicating which buttons are pressed, too. From these values you can decide what part of the program to execute.

As with most tutorials, this one too can be better explained with the use of a simple program that acts upon the user's interaction with the mouse and shows the bases of code needed to efficiently operate and control the mouse in your application.

IMPORTANT: It is mandatory that you set yourself in a graphic mode in your application. If you do not, the mouse coordinates will always return -1 for a value if the graphic mode is not set.

THE SAMPLE PROGRAM DESCRIPTION

For the sake of a demonstration program, things will be quite simple and will show 3 items at the top of the screen and depending on which one you click on, it will change the color of the screen. This should give you enough information to know how to work with the mouse in your application.

In FreeBasic, there's basically 2 commands that you need to worry about when it comes to the mouse. Here they are with their syntax explained as per the documentation.

Syntax

GETMOUSE x, y[, [wheel][, [buttons]]]

Description

GETMOUSE retrieves the mouse position and button status.

Mouse position is stored in X and Y when the function is called. If the mouse is not present or out of the program window, X and Y will be -1.

'wheel' is the mouse wheel counter. Rotating the wheel away from you makes it to decrease. If mouse is not present or out of the program window, wheel will be 0.

'buttons' stores the button status. On function termination, this will return a value where bit 0 is set if left mouse button is down; bit 1 is set if right mouse button is down; bit 2 is set if middle mouse button is down.

*GETMOUSE is for use in graphics modes, set using the SCREEN command.

Syntax

SETMOUSE x, y, visibility

Description

SETMOUSE will set the X,Y coordinates of the mouse pointer, as well as the visibility of the mouse.

Mouse position is set using the X and Y parameters.

The mouse will be visible if visibility is set to 1, and invisible if visibility is set to 0.

*SETMOUSE is intended for graphics modes initiated using the SCREEN command.

THE CODING BEGINS

Here are a set of constants that I declare at the beginning of the module, the rest of the programming example.

```
Const LEFTBUTTON = 1
Const MIDDLEBUTTON = 4
Const RIGHTBUTTON = 2
Const SHOWMOUSE = 1
Const HIDEMOUSE = 0
```

As a first step in this example, we will be declaring variables that we v course you don't have to declare your variables, but me I like to do so you're declaring your variables. To me that's good practice.

```
Dim CurrentX As Integer
Dim CurrentY As Integer
Dim MouseButton As Integer
Dim CanExit As Integer
Dim As String A,B,C
```

The idea here is to do everything within a loop so that we can also co that will exit when the "CanExit" variable is equal to 0. In the loop we'l values. (This part is extracted from the example provided in the GETM Don't forget to set your graphics mode as it is a must to get valid retur Screen 12 for our example.

```
Screen 12
CanExit = 1

Do While CanExit <> 0
    GetMouse CurrentX, CurrentY, , MouseButton
    If CurrentX < 0 Then
        Print "Mouse is out of context."
    Else
        If MouseButton And LEFTBUTTON Then A="L"
```

```

If MouseButton And MIDDLEBUTTON Then B="M"
If MouseButton And RIGHTBUTTON Then C="R"
Print Using "Mouse position: ###:### Buttons:
A="":B="":C=""
End If
Loop

```

This sample will basically continuously display information about which mouse button is pressed if any. The GETMOUSE statement based on CurrentX and CurrentY variables and the status of the mouse buttons. Statements will print L if the left button was pressed, M if the middle button was pressed.

For the next step, since we want to control a bit what's happening with the beginning of the program and control what happens with them afterwards, this could be replaced by a series of line commands or something. But that is outside the scope of this tutorial. So far, by getting rid of the loop, the loop should now look like this:

```

Screen 12
SetMouse 1, 1, 1
CanExit = 1
Locate 1,1
Print " | FIRST | SECOND | THIRD | EXIT | "
Do While CanExit <> 0
  Locate 1,1
  GetMouse CurrentX, CurrentY, , MouseButton
Loop

```

Basically we print the line that has " | FIRST | SECOND | THIRD | EXIT | " and then enter a loop that interrogates the mouse. Of course, right now nothing will happen for it. In our example, we'll add code that simply prints which option we'll print the Option and we'll exit the loop. We'll also add a print statement that we are truly outside the loop and therefore the program is ended. I am putting the whole source file here so you can cut and paste it easily.

```

Const LEFTBUTTON = 1
Const MIDDLEBUTTON = 4 ' UNUSED IN THIS DEMO

```

```

Const RIGHTBUTTON = 2 ' UNUSED IN THIS DEMO
Const SHOWMOUSE = 1
Const HIDEMOUSE = 0

Dim CurrentX As Integer
Dim CurrentY As Integer
Dim MouseButton As Integer
Dim CanExit As Integer

Screen 12
SetMouse 1, 1, SHOWMOUSE
CanExit = 1
Locate 1,1
Print " | FIRST | SECOND | THIRD | EXIT | "

Do
  GetMouse CurrentX, CurrentY, , MouseButton
  If MouseButton And LEFTBUTTON Then
    If CurrentY <= 12 Then
      If CurrentX >= 0 And CurrentX <=75 Then
        Locate 12, 1
        Print "First Option Selected ";
      ElseIf CurrentX >= 76 And CurrentX <= 147
        Locate 12, 1
        Print "Second Option Selected";
      ElseIf CurrentX >= 148 And CurrentX <=212
        Locate 12, 1
        Print "Third Option Selected ";
      ElseIf CurrentX >= 213 And CurrentX <=268
        Locate 12, 1
        Print "Last Option Selected ";
      Exit Do
    End If
  End If
End If
Loop While Inkey$ = ""

SetMouse 1, 1, HIDEMOUSE
Print

```

```
Print "AND NOW WE'RE OUT OF THE LOOP"  
Sleep
```

You can see the many IF statements in this last piece of code. The nu SCREEN 12 returned coordinates. They should work in all graphics r Console Graphics Window. Each if represents where the different opti used a graphics button routine you could simply use the same width a statements to know which button was clicked.

IN CONCLUSION

As you can see, using the mouse has been made very simple in Free command to draw your screens or you can use graphics command lik which way you choose to draw your screens with, the SETMOUSE ar and return the very same values. All you have to do is get that inform: them to do if they press a button, select an option, or even in the case character move towards the location where you clicked on the screen

As always, if you have any questions regarding this tutorial or any oth what we can do about solving your particular problem.

MystikShadows
Stéphane Richard
srichard@adaworld.com

Basic Input



Get Information into your Program.

Input is the life of any program. If you can't get something into your program anything out of it? What you will find here is the basics of how to get info into your program.

Here's a very basic program that will ask for your name:

```
'Create a place to put the user's name
Dim As String strMyName

' Ask for the user's name and store it in the string
Input "What is your name? ", strMyName

' Wait half a second
Sleep 500

' Show them their name
Print
Print "I now know your name is "; strMyName
Print

' Wait until someone presses a button before you exit
Print "Press any button to exit"
Sleep
```

INPUT is the easiest way to get information from someone. They just type it in when they are done.

What if you only want one keystroke? The easiest way is to use the **ASCII** value of a key that was pressed.

```
' Ask the user for input
```

```
Print "Press your favorite key:"

' Set a place to keep the ASCII value of the key
Dim As Integer strKeyPress

' Keep going until a key is pressed
Do
    strKeyPress = GetKey
Loop Until strKeyPress <> 0

' Show the key the user pressed
Print
Print "Your favorite key is: "; Chr(strKeyPress)

' Wait until someone presses a button before you exit
Print
Print "Press any button to exit"
Sleep
```

For more information check out the [User Input](#) Section.

TekRat

Contact me at: tekrat@2d.com

Dynamic Arrays



Hello, this page explains the proper use of dynamic arrays in FreeBASIC putting into the "Getting Started" tutorial page.

Arrays are neat; they can be used and resized throughout a program, will explain how to redimension a Dynamic Shared Array within a sub or

```
Declare Sub mySub ()

' as of 0.17, OPTION DYNAMIC and '$DYNAMIC are unneeded
' as you can see, both following ways are successful
Dim Shared myArray1() As UByte
ReDim Shared myArray2(0) As UByte

mySub

' because we shared the arrays, they are accessible
Print myArray1(5) ' will print 2
Print myArray2(6) ' will print 3

Sub mySub ()
    ' do NOT use "redim shared" within a sub or function
    ReDim myArray1(0 To 9) As UByte
    ReDim myArray2(0 To 9) As UByte
    myArray1(5) = 2
    myArray2(6) = 3
End Sub
```

Now, you may be wondering how you can redimension an array while using in fact, this only works if the first array dimension is the only one changing

```
' declare the dynamic array the cleaner way
ReDim Shared myArray(0 To 9, 0 To 9) As UByte
Dim As UByte x, y, i
```

```

' fill the array with values
For y = 0 To 9
    For x = 0 To 9
        i += 1
        myArray(x, y) = i
    Next x
Next y

' proves the values are good originally:
For y = 0 To 9
    For x = 0 To 9
        Print Using "##, "; myArray(x, y);
    Next x
    Print
Next y
Print
Print "Press a key..."
Sleep
Cls

' redimension the arrays
ReDim Preserve myArray(0 To 18, 0 To 12) As UByte

' the values have not been preserved properly!
For y = 0 To 9
    For x = 0 To 9
        Print Using "##, "; myArray(x, y);
    Next x
    Print
Next y

Sleep
End

```

Try it out! You can see that it does not work properly. This is because on PRESERVE to work properly.

There is a workaround, which I will post later, after I edit it in order to ma
moment, get creative ;)

Introduction

This tutorial is aimed at people who want to know more about the new feature being referred to as 'types as objects', and 'that OOP stuff'. It aims to warn people who don't really understand it yet, but want to learn. A data type, like a struct in C, or a record in Pascal. Here's just a short sample

```
Type person_info
  first_name As String
  last_name As String
  house_number As Integer
  street_name As String
  town As String
End Type
```

In this usage it's used as a kind of container for related data; in this example an address book. With the new features, however, it can be used more like much more than contain just simple fields of data. It becomes a way to create objects that makes object oriented programming much simpler. We will now look at the

Property

We'll start by looking at property. When you add a property to a Type, you get a member, but what happens, is instead of just getting or setting a variable. Take a look at this example:

```
Type bar
  Declare Property x() As Integer
  Declare Property x(ByVal n As Integer)
  p_x As Integer
End Type

Property bar.x() As Integer
```

```

    Print "bar.x()"
    Property = p_x
End Property

Property bar.x(ByVal n As Integer)
    Print "bar.x(ByVal n As Integer)"
    p_x = n
End Property

' ---

Dim foo As bar

foo.x = 5
Print foo.x

```

We include in our `Type` some declarations for a `Property`; they are very simple declarations. The first one declares a getter, the second a setter. The `p_x` member.

Next we write the code for the properties; again, the syntax is very similar to the way we return a value: instead of `Function = value`, we do `Property = value` as well. Also note that you can refer to the member directly as `p_x`; you can also use `this.p_x` in an example `this.p_x = n`; using `this` isn't usually needed, but it can help in some cases.

Then follows some testing code; this shows how we can use the `Property` member. When you run the program it will also print to screen to show that the `Property` works.

Now this code is fairly trivial, but as you get used to the idea you'll see it can be very useful. Imagine as an example you are writing a GUI, and the `Type` represents a `button`. You could do `button.text = "Hello World!"`, and make the property code update the `text` property. Maybe you are using the `Type` to maintain some kind of list; you could do `list.Add item` code in your property to make the list larger.

Constructor/Destructor

Constructors are functions that are called when the `Type` gets created - \

Destructor is a function that gets called when the Type goes out of scope, for a Type in the main code, or when a function ends, for a local Type expanded from the last.

```
Type bar
  Declare Constructor()
  Declare Destructor()
  Declare Property x() As Integer
  Declare Property x(ByVal n As Integer)
  p_x As Integer Ptr
End Type

Constructor bar()
  Print "Constructor bar()"
  p_x = Allocate(SizeOf(Integer))
  *p_x = 10
End Constructor

Destructor bar()
  Print "Destructor bar()"
  Deallocate(p_x)
End Destructor

Property bar.x() As Integer
  Print "bar.x()"
  Property = *p_x
End Property

Property bar.x(ByVal n As Integer)
  Print "bar.x(ByVal n As Integer)"
  *p_x = n
End Property

' ---

Dim foo As bar

Print foo.x
```

```
foo.x = 5
Print foo.x
```

Again the syntax is somewhat similar to normal functions. Note that this ptr. The constructor then Allocates the memory for this when foo is created it De-Allocates this memory once it is destroyed. So you can use constructor for you, then clean up once its finished with. Again a trivial example, kind of list, and having it set the list up for you, and clean it up when it's finished.

Methods

You can also have regular subs and Functions inside your Type; in some methods. We'll carry on our example:

```
Type bar
  Declare Constructor()
  Declare Destructor()
  Declare Property x() As Integer
  Declare Property x(ByVal n As Integer)
  Declare Sub Mul5()
  Declare Function Addr() As Integer Ptr
  p_x As Integer Ptr
End Type

Constructor bar()
  Print "Constructor bar()"
  p_x = Allocate(SizeOf(Integer))
  *p_x = 10
End Constructor

Destructor bar()
  Print "Destructor bar()"
  Deallocate(p_x)
End Destructor

Property bar.x() As Integer
```

```

Print "bar.x()"
Property = *p_x
End Property

Property bar.x(ByVal n As Integer)
Print "bar.x(ByVal n As Integer)"
*p_x = n
End Property

Sub bar.mul5()
*p_x *= 5
End Sub

Function bar.Addr() As Integer Ptr
Function = p_x
End Function

' ---

Dim foo As bar

Print foo.x
foo.x = 5
Print foo.x
foo.mul5()
Print foo.x
Print "address p_x points to", foo.Addr()

```

So this time we added a sub, that multiplies the integer pointed to by `p_x` memory address that the pointer holds.

Private/Public

By default all of the members of the `bar` type are public; that means that However, sometimes you might want to make them private. Take for example currently do `Print *foo.p_x`, and it will allow us to print the value it points to so that only the members of the `bar` type (the constructor, destructor, pro

That way we can make sure we only deal with `p_x` by the ways we choose: `DeAllocate(foo.p_x)` in our main code, then when the destructor runs, it's 'double free'. Change the `Type` declaration as follows:

```
Type bar
  Declare Constructor()
  Declare Destructor()
  Declare Property x() As Integer
  Declare Property x(ByVal n As Integer)
  Declare Sub Mul5()
  Declare Function Addr() As Integer Ptr
Private:
  p_x As Integer Ptr
End Type
```

Now try adding `Print *foo.p_x` to the main code and compile it. You'll get an `Illegal member access, found 'p_x' in 'Print *foo.p_x'`, showing the compiler enforcing the fact we made `p_x` private. When you use `private:` or `public:` statements, they must follow the rule. Here's a rather pointless example just to show

```
Type bar
Private:
  a As Integer
  b As Integer
Public:
  c As Integer
  d As Integer
Private:
  e As Integer
End Type
```

In the above type, the members `a`, `b`, and `e` are private; `c` and `d` are public.

Operator overloading

Operator overloading is a way of telling the compiler what to do in the case of a certain kind of operation involving our `Type`. Take this example:

```
Type bar
  n As Integer
End Type

Dim As bar x, y, z

z = x + y
```

Now normally the compiler will throw an error when it sees this, as it has no idea how to add two `bar` types, but we can define what we want to happen. Here's how:

```
Type bar
  n As Integer
End Type

Operator +(ByRef lhs As bar, ByRef rhs As bar) As bar
  Operator = Type(lhs.n + rhs.n)
End Operator

Dim As bar x, y, z

x.n = 5
y.n = 10
z = x + y
Print z.n
```

In this code, I use `lhs` and `rhs` to refer to the left and right hand side of the operation type(`lhs.n + rhs.n`); this builds the `Type` that will be returned.

```
Type bar
  x As Integer
  y As Integer
  z As Integer
End Type
```

Then you would build it like `type(xpart, ypart, zpart)`.

Most or all operators can be overloaded, and most of them are binary or like the + example above. Some are unary ops having only a right hand would be done like `'Operator Not(ByRef rhs As bar) As bar'`.

There are some special cases where they have to be declared inside the operators and casts.

Assignment operators are things like `+= -= mod=` etc, and also `Let`. `Let` is like:

```
Dim As bar foo
Dim As Integer x
foo = x
```

And casts are kind of the reverse; they are used when you cast to another

```
Dim As bar foo
Dim As Integer x
x = foo
```

Here's a short example using `Let` and `Cast`:

```

Type bar
  n As Integer
  Declare Operator Let(ByRef rhs As Integer)
  Declare Operator Let(ByRef rhs As String)
  Declare Operator Cast() As String
End Type

Operator bar.Let(ByRef rhs As Integer)
  n = rhs
End Operator

Operator bar.Let(ByRef rhs As String)
  n = Val(rhs)
End Operator

Operator bar.Cast() As String
  Operator = Str(n)
End Operator

Operator +(ByRef lhs As bar, ByRef rhs As bar) As bar
  Operator = Type(lhs.n + rhs.n)
End Operator

Dim As bar x, y, z

x = 5
y = "10"
z = x + y
Print z

```

You need to have separate lets and casts for each data type you want to declaring within the type are known as non-static, and the ones that don't are known as static. The technical reason for this; the non-static ones need to know which instance of the type they are operating on (in the example above, we would say that x is an instance of bar) of the type they are operating on, which is accomplished by a hidden 'this' reference. This hidden 'this' reference is used by operators and methods to know which instance of the type the call refers to.

here's a list of the ones that currently can be:

Assignment ops:

let, +=, -=, *=, /=, \=, mod=, shl=, shr=, and=, or=, xor=, imp=, eqv=, ^=

Unary ops:

-, not, @, *, ->

Binary ops:

+, -, *, /, \, mod, shl, shr, and, or, xor, imp, eqv, ^, =, <>, <, >, <=, >=

Overloaded Constructors/Methods

As with normal functions, our `Type`'s constructor and methods can be overloaded. This provides a way to specify details on how the instance should be constructed.

```
Type bar
  Declare Constructor()
  Declare Constructor(ByVal initial_val As Integer)
  x As Integer
End Type

Constructor bar()
  x = 10
End Constructor

Constructor bar(ByVal initial_val As Integer)
  x = initial_val
End Constructor

Dim foo As bar
Print foo.x

Dim baz As bar = bar(25)
Print baz.x
```

The first constructor, that had no arguments, is known as the default constructor and provides an initial value of 10. However, we have also specified another constructor

the way we ask for this to be called `Dim baz As bar = bar(25)`. You can and then you will always have to specify the initial value using the `const` can't have an overloaded destructor, because there's no way to manually

Overloaded methods are very similar:

```
Type bar
  Declare Sub foo()
  Declare Sub foo(ByVal some_value As Integer)
  Declare Sub foo(ByRef some_value As String, ByVal
  x As Integer)
End Type
```

They work just the same as normal overloaded functions.

Closing

I hope this tutorial has been useful for you, although there are still a few far, it shouldn't be too hard for you to pick them up. There is some more on the forums, and also in part 2 of this tutorial, available here - [Beginn 2\)](#)

More reading

- [Property](#)
- [Constructor](#)
- [Destructor](#)
- [Operator](#)
- [This](#)
- [Type](#)
- [Types as Objects](#)
- [Public:](#)
- [Private:](#)
- [Protected:](#)

Introduction.

Welcome to the second part of the tutorial, In this part I assume that you have read the examples, and experimented with some tests of your own. I'll now cover some topics that were not included in Part 1.

Indexed property.

An indexed property is a property that behaves like an array, except that when you access the property, a function gets called when you access it. I'll start with a very simple example.

```
Type foo
  Declare Property bar(ByVal index As Integer, ByVal value As Integer) As Integer
  Declare Property bar(ByVal index As Integer) As Integer
  dummy As Integer
End Type

Property foo.bar(ByVal index As Integer, ByVal value As Integer) As Integer
  Print "Property set, index=" & index & ", value=" & value
End Property

Property foo.bar(ByVal index As Integer) As Integer
  Print "Property get, index=" & index
  Property = 0
End Property

Dim baz As foo

baz.bar(0) = 42
Print baz.bar(0)
```

As you can see, the declaration for our indexed property is very similar to a regular property, but we add an argument for the index. I include a dummy integer member, but

least one data member. As you can see, the property is then used with (the zeroth index, just the same as we would for an ordinary array. Now I have a useful example, and I will describe it:

```
Type foo
  Declare Constructor(ByVal num_elements As Integer)
  Declare Destructor()
  Declare Property bar(ByVal index As Integer, ByVal value As Integer)
  Declare Property bar(ByVal index As Integer) As Integer
Private:
  x As Integer Ptr
  size As Integer
End Type

Constructor foo(ByVal num_elements As Integer)
  x = CAllocate(num_elements * SizeOf(Integer))
  size = num_elements
End Constructor

Destructor foo()
  Deallocate(x)
End Destructor

Property foo.bar(ByVal index As Integer, ByVal value As Integer)
  If (index >= 0) And (index < size) Then
    x[index] = value
  Else
    Error 6
  End If
End Property

Property foo.bar(ByVal index As Integer) As Integer
  If (index >= 0) And (index < size) Then
    Property = x[index]
  Else
    Error 6
  End If
End Property
```

```

Dim baz As foo = foo(10)

baz.bar(1) = 42
Print baz.bar(1)

```

This time, I've added a constructor and destructor, which will allocate an array, x, with the number of elements specified in the constructor. Then when invoked, I check if the index is within the bounds of the array, if it is then set. If the index specified is out of bounds, then 'Error 6' occurs, which is VB's 'out of bounds error', you could replace this with your own error handler by changing the code 'baz.bar(1) = 42' to 'baz.bar(10) = 42', and you'll see 10 elements (index 0-9)

Copy constructor.

A copy constructor is a special type of constructor, that is used to make a copy of an object. When you write code like this:

```

Type foo
...
End Type

Dim As foo a
Dim As foo b = a

```

What happens is FreeBASIC automatically generates hidden code to copy the object. This is the default copy constructor, and simply copies the data fields (members). If you want your own copy constructor, here's just a brief snippet to show how we declare it:

```

Type foo
  Declare Constructor(ByRef obj As foo)
  ...
End Type

```

This will come in very useful for a reason I will now explain.

Deep/Shallow copy.

In that previous example, where we did the code 'Dim As foo b = a', that copy, it just simply copied the data fields across, however sometimes this one of the members is a pointer, what will happen is that the address that is copied across, so both objects will point to the same memory. An example of this is shown below:

```
Type foo
  x As Integer Ptr
End Type

Dim As foo a

a.x = Allocate(SizeOf(Integer))
*a.x = 42

Dim As foo b = a

Print *a.x, *b.x

*a.x = 420

Print *a.x, *b.x

Deallocate(a.x)
```

As you see, because they both point to the same memory, changing one in the previous section on the copy constructor, FreeBASIC creates the default. This is also true if we do an assignment like:

```
Dim As foo a, b
```

```
b = a
```

In this case also, FreeBASIC creates a default assignment operator (Let) in order to do deep copies, we need to define a copy constructor, and an overloaded operator to accept our type. Here's an example using them.

```
Type foo
  Declare Constructor()
  Declare Constructor(ByRef obj As foo)
  Declare Destructor()
  Declare Operator Let(ByRef obj As foo)
  x As Integer Ptr
End Type

Constructor foo()
  Print "Default ctor"
  x = CAllocate(SizeOf(Integer))
End Constructor

Constructor foo(ByRef obj As foo)
  Print "Copy ctor"
  x = CAllocate(SizeOf(Integer))
  *x = *obj.x
End Constructor

Destructor foo()
  Print "dtor"
  Deallocate(x)
End Destructor

Operator foo.Let(ByRef obj As foo)
  Print "Let"
  *x = *obj.x
End Operator
```

```

Dim As foo a

*a.x = 42

Dim As foo b = a 'Uses the copy constructor

Print *a.x, *b.x

*a.x = 420

Print *a.x, *b.x

```

As you can see, the copy constructor gets called on the line 'Dim As foo b = a' to allocate some memory, and copy the data in the new copy constructor, so that without it affecting the other. If we change the main code as follows:

```

Dim As foo a, b

*a.x = 42
b = a 'The assignment operator (Let) gets used th

Print *a.x, *b.x

*a.x = 420

Print *a.x, *b.x

```

Then this time the assignment operator is used. Note that in the assignment we need to allocate any memory because it has already been allocated in the first object, so we need to copy the data across. The line '*x = *obj.x' performs this copy. If we had a dynamic memory array, then we would need to reallocate the array to the size to fit the data being copied. Here's a more advanced version just to

```

Type foo

```

```

Declare Constructor(ByVal num_elements As Integer)
Declare Constructor(ByRef obj As foo)
Declare Destructor()
Declare Operator Let(ByRef obj As foo)
x As Integer Ptr
size As Integer
End Type

Constructor foo(ByVal num_elements As Integer)
Print "Default ctor"
x = CAllocate(SizeOf(Integer) * num_elements)
size = num_elements
End Constructor

Constructor foo(ByRef obj As foo)
Print "Copy ctor"
x = CAllocate(SizeOf(Integer) * obj.size)
size = obj.size
For i As Integer = 0 To size - 1
    x[i] = obj.x[i]
Next i
End Constructor

Destructor foo()
Print "dtor"
Deallocate(x)
End Destructor

Operator foo.Let(ByRef obj As foo)
Print "Let"
x = Reallocate(x, SizeOf(Integer) * obj.size)
size = obj.size
For i As Integer = 0 To size - 1
    x[i] = obj.x[i]
Next i
End Operator

Dim As foo a = foo(5)

```

```

a.x[0] = 42
a.x[1] = 420

Dim As foo b = a 'Uses the copy constructor

Print a.x[0], a.x[1], b.x[0], b.x[1]

b.x[0] = 10
b.x[1] = 20

Print a.x[0], a.x[1], b.x[0], b.x[1]

b = a ' Now using the assignment operator

Print a.x[0], a.x[1], b.x[0], b.x[1]

```

This may seem quite complex at first, it's worth just reading through it a few examples, it's not too tricky once you're used to it.

Passing objects to functions ByVal

The idea of deep and shallow copies also applies to passing an object to a function. If you pass a reference to an object (ByRef), you can modify the object, and the changes will be visible outside of the function. However, you can also pass by value, which will mean you can modify it inside the function, but the changes will not be visible outside. When an object is passed by value to a function, that object has a copy constructor, then this is invoked, if it doesn't, then a shallow copy is performed. Once the function ends, the object's destructor is called.

New/Delete

New and delete are special operators for dynamically allocating memory. New is used with dynamic memory, it is used with pointers. In all the examples I've shown, I create our objects, this will create them on the stack, but by using new we can allow more flexibility, just like using Allocate/DeAllocate with pointers. An important thing about new, is that you don't need to check if the pointer is null if you did allocate. If new fails, it causes an exception, which will end the program. In FreeBASIC, it is likely that some kind of try..catch mechanism will be created.

handling, but as of the time of writing, this is not yet implemented.

There are two different varieties of the new/delete. The first type, create: for example:

```
Dim As Integer Ptr foo = New Integer

*foo = 1
Print *foo

Delete foo
```

This will create a new Integer, then destroy it when we call delete. Remember dynamic memory. For simple data types you can also specify a default value after the data type, ie:

```
Dim As Integer Ptr foo = New Integer(42)

Print *foo

Delete foo
```

This also works for UDT's with just simple data fields:

```
Type foo
  x As Integer
  y As Integer
End Type

Dim As foo Ptr bar = New foo(1, 2)

Print bar->x, bar->y
```

```
Delete bar
```

This initialization won't work for more complex types involving constructors. A useful feature is that when using new/delete with objects, it also calls the destructor. The following example:

```
Type foo
  Declare Constructor()
  Declare Destructor()
  x As Integer
  y As Integer
End Type

Constructor foo()
  Print "ctor"
End Constructor

Destructor foo()
  Print "dctor"
End Destructor

Dim As foo Ptr bar = New foo

Delete bar
```

You will see that the constructor and destructor for the object are called.

The second type of new/delete is for creating arrays, this time the number of elements and datatype in square brackets '[]'. When using the array version, you must use 'delete', so that FreeBASIC knows you are deleting an array, here is a simple example:

```
Dim As Integer Ptr foo = New Integer[20]
```

```
foo[1] = 1
Print foo[1]

Delete[] foo
```

This will create a dynamic array, with 20 Integer elements. It should be r Allocate, which takes the number of bytes as its argument; using new, y elements. The array method works just the same for objects:

```
Type foo
  Declare Constructor()
  Declare Destructor()
  x As Integer
  y As Integer
End Type

Constructor foo()
  Print "ctor"
End Constructor

Destructor foo()
  Print "dtor"
End Destructor

Dim As foo Ptr bar = New foo[3]

Delete[] bar
```

When you run this code, you will see that three constructor/destructor p created an array of three instances of foo.

You must remember to call Delete, or Delete[] for any memory allocated memory leak, just like the way you must rememeber to call DeAllocate f the Allocate function.

Name Mangling

Name mangling, also known as name decoration, is something that happens at a lower level, and as such is not essential to know about. The reason for the problems that are involved with more than one function sharing the same name is that functions are overloaded, or are part of a type. Take for example the overloads below:

```
Sub foo Overload ()  
  
End Sub  
  
Sub foo(ByVal i As Integer)  
  
End Sub
```

If we didn't have name mangling, then both might be known at a lower level, causing a name clash, so they have to be decorated in order to know which one is used. For the first sub, the compiler actually creates a sub called `_Z3FOO` and for the second a sub called `_Z3FOOi`. The compiler then remembers these, and chooses the correct one depending on how you call it, for example 'foo()' will actually call `_Z3FOO` and 'foo(1)' will call `_Z3FOOi`. We can spot something from this, that the 'v' stands for void (or value) and the 'i' stands for integer. The full details of name mangling are quite complex, and vary between compilers. GNU compilers use a different name mangling scheme to Microsoft compilers, and different schemes as well. The main thing we need to know, is that FreeBASIC (Application binary interface), meaning that any overloaded functions, or member functions, are compatible with other compilers using the same scheme. This is an unfortunate fact, really a FreeBASIC problem, it is common of all the compilers that use a name mangling scheme, that all the compiler authors agreed on a common name mangling scheme, that would cause incompatibility.

Implicit this

This again is not necessary to know about mostly, it's something that happens at a lower level. When you call a member function of an object, what actually happens is that a parameter is passed, so that the function knows which instance of the object it is being called on.

also true for the property/constructor/destructor/operator members. If we

```
Type foo
  Declare Sub bar(ByVal n As Integer)
    x As Integer
End Type

Sub foo.bar(ByVal n As Integer)
  x = n
End Sub

Dim baz As foo
baz.bar(5)
```

What actually happens behind the scenes is something essentially equiv

```
Type foo
  x As Integer
End Type

Sub foo_bar(ByRef _this As foo, ByVal n As Integer)
  _this.x = n
End Sub

Dim baz As foo
foo_bar(baz, 5)
```

This method using an explicit 'this' is often used in languages that do no OOP is really just a set of concepts, that can be mostly coded in almost more difficult to implement, such as constructors, you would have to exp function. For some things such as private/public distinction, it is even mc because the compiler does not know to enforce them. The reason for ac language is to hide a lot of this, and add syntactic sugar to make it simpl use, such as the way we can use properties as if they were ordinary dat

functions, which is what they really are.

Hints for debugging/profiling

When using GDB or other debuggers, and the gprof profiling tool, the int syntax, and all your variable names and other symbols are shown in up overview to help you understand how these are shown:

Here's an example type:

```
Type bar
  Declare Constructor()
  Declare Constructor(ByRef obj As bar)
  Declare Constructor(ByVal n As Integer)
  Declare Destructor()
  Declare Operator Cast() As Any Ptr
  Declare Operator Let(ByVal n As Integer)
  Declare Property foo(ByVal n As Integer)
  Declare Property foo() As Integer
  member As Any Ptr
End Type
```

When using GDB, these will be shown as follows (note in C++ they use is known as the scope resolution operator):

BAR::BAR() - The default constructor

BAR::BAR(BAR&) - The copy constructor (& in C++ means a reference,

BAR::BAR(int) - The constructor taking an integer argument (note there ByVal, as this is the default passing method in C/C++)

BAR::~~BAR() - The destructor

BAR::operator void*() - A cast to Any ptr (void is similar to Any, * means

BAR::operator=(int) - The assignment operator (Let), denoted by '=', in equality testing.

BAR::FOO(int) - Property foo setter, taking an integer argument

BAR::FOO() - Property foo getter

Member sub/functions are shown in the same way as properties, indexes same also, just with the extra argument for the index.

Here is how the FB data types will be shown:

Any ptr - void *
ZString ptr - char *
String - FBSTRING
byte - signed char
ubyte - bool
short - short
ushort - unsigned short
integer - int
uinteger - unsigned int
longint - long long
ulongint - unsigned long long

I hope that helps you get started with understanding how things are done. experimentation will always help.

More reading

<http://www.freebasic.net/wiki/wikka.php?wakka=KeyPgOpNew>
<http://www.freebasic.net/wiki/wikka.php?wakka=KeyPgOpDelete>
http://en.wikipedia.org/wiki/Copy_constructor
http://en.wikipedia.org/wiki/Object_copy
http://en.wikipedia.org/wiki/Name_mangling

Introduction to Variable Scope



Written by *rdc*

Variable Scope

Scope refers to the visibility of a variable, where you can access a variable in a program. Before you can understand the different levels of scope, you need to understand the structure of a program in FreeBasic.

Program Structure

A complete program is composed of one or more .bas files, called modules. A module can contain both module level code, and code contained within subroutines and functions. Module level code is code that is *not* contained within a subroutine or function. The following snippet illustrates the various parts of a module.

```
Dim aInt As Integer 'Variable declared at module level

Sub DoSomething
    Dim aInt As Integer 'Variable declared at sub level
    ... 'This code is local to sub
End Sub

Function DoSomethingElse() As Integer
    Dim aInt As Integer 'Variable declared at function level
    ... 'This code is local to function
End Function

'Module level code
aInt = 5
DoSomething
aInt = DoSomethingElse()
```

Local Variables

If you define a variable at the module level (and not using `shared`), the variable has local module level scope. It is visible to the module level code, but not to subroutines or functions within the module. In the example above the module-level variable `aInt` is only visible to the module level code.

Variables defined within a subroutine or function are local to the subroutine or function and are not visible to module level code or any other subroutine or function.

Variables Defined Within Control Structures

Variables that are defined within `If`, `For-Next`, `While-Wend` and `Do-Loop` are local to the control structure block code. That is, they are not visible outside the bounds of the begin and end of the control block, just like a variable declared within a subroutine or function.

Shared Variables

In the example, if you wanted `aInt` to be visible within the subroutine or function, you would need to declare the variable as `shared` and then not declare a variable with the same name within any subroutine, function or control block. A `shared` variable is visible to module level code, subroutine or function level code and within control structure blocks.

Scope Conflicts

In the code snippet above, if `aInt` were declared as `shared`, and each subroutine or function declared `aInt`, there would be a scope conflict, since the same name is used for different levels of scope.

The compiler resolves this by taking the current scope into account and using the variable within that scope. Since subroutines and functions have a lower scope than the module, `aInt` would refer to the variable declared within the subroutine or function, and not the one declared at the module level, even though it is declared as a `shared` variable.

Multiple Modules

Scope is limited to a single module, that is a single .bas file. However, it is necessary to extend the scope from one module to another. You would use the `Common` statement when you declare a variable that needs to be shared across multiple modules.

Each module must have the same `Common` declaration in order for the variables to match up. If you declare a variable in `module1` as `Integer` then `module2` must also have `Common aInt as Integer`. Without the `Common` declaration `aInt` would not be visible within `module2`.

You can add the `shared` attribute to `Common`, that is `Common Shared` to not only extend scope to multiple modules, but to extend scope within a module. `Common Shared` operates the same as `shared` within a single module. As with `Common`, you have matching declarations in each module that needs access to the variable.

Scope...End Scope

You can create a temporary scope block by using the `Scope`, `End Scope` block. This scope block is very useful when creating multi-line macros where you may create some temporary working variables but do not want to introduce them into the program. The following snippet illustrates how to create a scope block.

```
Scope
  Dim tmp As Integer
  ... 'Some code
End Scope
```

The scope of any variable created within a scope block is limited to the block. However, the scope block inherits the visibility of the surrounding scope. Variables created at the same scope as the scope block are visible within the scope block.

For example, if you have `aInt` which is at module level scope, and the scope block also has a variable `aInt` at module level scope, then `aInt` would be visible inside the scope block. However, there is a scope conflict, in which case the variable inside the scope block would override the variable with the same name outside the scope block.

Variable Lifetime

Not only does scope set the visibility of a variable, it also determines the variable's lifetime. A variable goes through several stages in its lifetime; creation, access and destruction. When this occurs depends on the scope of a variable where the variable has been defined within the program.

Module Level Variables

Module level variables exist for the life of a program, since they are declared in the main body of the program. Module level code is the main executing program, and terminates when the program ends.

Subroutine and Function Level Variables

Variables declared within a subroutine and function exist as long as they are used within the body of the subroutine and function. On entering the sub/function, the variable is created, initialized and can be accessed within the sub/function. Once the sub/function exits, the variable is destroyed.

Static Variables

One exception to the declared sub/function variable is the `static` variable. `static` variables maintain their value between calls to the subroutine or function over a module level lifespan.

Control Block Variables

Variables declared within a control block, such as a For-Next, exist as long as the control block is executing. Upon leaving the control block, the variables are destroyed.

Scope...End Scope Variables

Variables declared within a scope block exist as long as the scope block is executing. Once the program leaves the scope block, any variables created within the scope block are destroyed.

rdc



Rick Clark aka rdc
rickclark58@yahoo.com
<http://rickclark58.bravehost.com/>

Introduction To Arrays



Written by *rdc*

Arrays are probably the single most useful programming construct that is used in a programming solution involve data arranged in tabular format, and array manipulation is a crucial skill in becoming a competent programmer.

Arrays are contiguous memory segments of a single or composite data type. An array can have one or more rows, and each row can have one or more columns. FreeBasic uses the row-major scheme for arrays, which means that the first dimension is the row. FreeBasic supports up to eight dimensions in an array.

One-Dimensional Arrays

An array with a single row is called a one-dimensional array. If an array is declared with a single row, only the number of columns in the row. Since an array requires a declaration, only the number of columns in the row. Since an array requires a declaration, only the number of columns in the row. The following code snippets create a single-dimension integer array using two different methods.

```
Dim myArray(10) As Integer  
  
Dim myArray(1 To 10) As Integer
```

The first method will define an array with a single row and 11 columns, while the second method defines the lower and upper bounds using the `To` keyword. Here the index starts at 1 and ends at 10.

One-Dimensional Array Indexes

You access each element of an array using an index value. In the case of a one-dimensional array, the index value is the column number. The default row is the first row. The format is to use the array variable, with the index value in parentheses.

```
myArray(5) = 7
```

This would set column 5 in the array to 7.

```
myInt = myArray(5)
```

This will set the value of `myInt` to the current value of column 5 in `myArra`

Two-Dimensional Arrays

A two-dimensional array is an array that has more than one row, along with a defined number of rows, where each row has a defined number of columns.

```
Dim myArray(2, 10) As Integer
```

The first dimension defines the number of rows in the array, while the second dimension defines the number of columns. In the example, the array has 3 rows, numbered 0 to 2, and each row has 11 columns.

You can also define the lower and upper bounds of the array.

```
Dim myArray(1 To 2, 1 To 10) As Integer
```

This definition would set the number of rows to 2, numbered 1 to 2 and the number of columns to 10, numbered 1 to 10.

Two-Dimensional Array Indexes

To access the array elements of a two-dimensional array, you would use the `ArrayName(row, column)` syntax, where `row` selects a row within the array and `column` selects a column within that row.

```
myArray(1, 5) = 7
```

This code would set column 5 in row 1 to 7.

```
myInt = myArray(1, 5)
```

This code would set myInt to the current value contained within column 1 in row 5.

Multi-Dimensional Arrays

For arrays of three or more dimensions, you would use the same format for each dimension. For a three-dimensional array, the first dimension would be the row, the second dimension would be the column, and the third dimension would be the depth axis.

For example, to define a cube in space, you would use the y,x,z format, where y defines the row axis, x defines the column axis, and z defines the depth axis. To create an array in this format you could define

```
Dim myCube(y, x, z) As Integer.
```

myCube(10, 10, 10) would create a cube with 11 vertical units, 0 to 10, 11 horizontal units, 0 to 10, and 11 depth units, 0 to 10. To find the center of the cube, you would use iCenter = myCube(5, 5, 5).

You will probably never need to use arrays of more than three dimensions. However, if you need to use higher-dimensional arrays, the same principles apply.

Dynamic Arrays

The arrays described above are static arrays; the array size cannot change during execution. Dynamic arrays are useful for creating arrays that can change size during execution.

Static arrays, the arrays described above, are kept on the heap, but the dynamic array compiler dynamically allocates memory for the array based on the requested size.

You specify a dynamic array by using the `ReDim` keyword.

```
ReDim myArray(1 To 5, 1 To 5) As Integer
```

If you don't know the needed array bounds at the start of the program e>

```
Dim myArray() As Integer
```

In this case the compiler sets a default value of 0 for the array size. You bounds.

ReDim and ReDim Preserve

Dynamic arrays can change sizes during execution. `ReDim` will clear the c will keep intact the existing contents, unless the array size is smaller tha

Array Functions

There are a number of **functions** that you can use with arrays.

Arrays of Composite Types

Type definitions allow you to group related data into a single entity, and c data. Arrays of types allow you create multiple instances of a type defini of this usage may be an inventory system for your RPG, a series of docu random access database.

You create arrays of types just as you would with any of the intrinsic data

```
Type myPoint
```

```

    row As Integer
    col As Integer
End Type

Type myLine
    p1 As myPoint
    p2 As myPoint
    char As String * 1
End Type

Dim myLineSet (1 To 3) As myLine

```

The code defines a set of 3 lines, with endpoints p1 and p2, where each using a combination of array index and dot operator.

```

myLineSet(1).p1.row = 1
myLineSet(1).p1.col = 1
myLineSet(1).p2.row = 10
myLineSet(1).p2.col = 10
myLineSet(1).char = Chr(219)

```

Arrays in Types

Not only can you create an array of a composite type, you can have an array more efficiently by replacing p1 and p2 with an array.

```

Type myPoint
    row As Integer
    col As Integer
End Type

Type myLine
    pts(1 To 2) As myPoint

```

```

    char As String * 1
End Type

Dim myLineSet (1 To 3) As myLine

```

Here `pts` is an array of `myPoint`. To access this structure you would use a

```

myLineSet(1).pts(1).row = 1
myLineSet(1).pts(1).col = 1
myLineSet(1).pts(2).row = 10
myLineSet(1).pts(2).col = 10
myLineSet(1).char = Chr(219)

```

`myLineSet` is an array, so you use an index value. `pts` is an element of the array, so you use an index to select each `pts` array element. Row and column operators.

Using an array for the endpoints enables you to easily extend the line definition. The code snippet shows one possible definition.

```

Type myObj
    objid As Integer
    Union
        myLine(1 To 2) As myPoint
        myTriangle(1 To 3) As myPoint
        mySquare(1 To 4) As myPoint
    End Union
    char As String * 1
End Type

```

The `objid` field would indicate which type of object is contained within the structure. A 1 may indicate a line, a 2 may indicate a triangle and a 3 may indicate a square. Since the definition defines a single

memory usage.

To print the object to the screen, you would examine the objid and then print the number of lines that correspond to the type of object.

One further enhancement you can make to this program is to add a function that corresponds to the type of object being printed. Using this technique will ease the process of adding new objects to the type definition.

For example, if you needed to be able to describe a cube, your type definition would be able to print a cube by simply adding a few lines

Array Initialization

You can initialize an array with values when using the `Dim` statement in a type definition. The following code snippet illustrates the syntax using a

```
Dim aArray(1 To 5) As Integer => {1, 2, 3, 4, 5}
```

This code snippet dimensions an integer array with 5 elements, then the assignment operator, `=>` tells the compiler that the list following the `Dim` statement should

You can also dimension multidimensional arrays in the same manner, by the code snippet illustrates.

```
Dim bArray(1 To 2, 1 To 5) As Integer => {{1, 2, 3,
```

In this example, the first block, `{1, 2, 3, 4, 5}`, corresponds to row 1, and so on. Remember that FreeBasic arrays are row-major, so the row is specified first. Be sure that the number of elements defined will fit into the array.

Type Array Initialization

Not only can you initialize an array of simple data types, you can also initialize an array that contains an array as an element of the type.

```
Type aType
  a As Integer
  b As Byte
  c(1 To 2) As String * 10
End Type

Dim As aType myType(1 To 2) => { (1234, 12, {"Hello'
```

The curly brackets signify that this is an array initialization, while the parentheses signify that this is a multidimensional array, then you would need to wrap each row in { and }

Using the -exx Compiler Switch

The -exx compiler switch will enable error and bounds checking within your program, the compiler will generate an "out of bounds" error while the program is running.

This is a great help in debugging your program, and finding problems as you make assignments, so it is quite useful when working with pointers as well.

Using -exx does add quite a bit of additional code to your program, so it is not recommended for a production program without the -exx switch.

Introduction to the Type Def



Written by *rdc*

There are times when creating a program that you may want to define a such as a personnel record, or an enemy in a game. While you can do this with primitive data types, it is hard to manage within a program. Composite data types group together related data items into a single structure that can be manipulated. FreeBASIC offers two composite data types, the `Type` and `Union`.

Types

FreeBASIC allows you to group several data types into a unified structure. A `Type` definition which you can use to describe these aggregate data structures.

The basic structure of a type definition is:

```
Type typename
    Var definition
    Var definition
    ...
End Type
```

The `Type-End Type` block defines the scope of the definition. You define the type structure in the same manner as using the `Dim` keyword, without using the `As` keyword. The code snippet shows how to build an employee type.

```
Type EmployeeType
    fname As String * 10
    lname As String * 10
    empid As Integer
    dept As Integer
End Type
```

You can use any of the supported data types as data elements, including type definitions. When you create the type definition, such as in the example just creating a template for the compiler. In order to use the type definition a variable of the type, as the following code snippet illustrates.

```
Dim Employee As EmployeeType
```

Once you have created a variable of the type, you can access each element using the dot notation *var_name.field_name*.

Using the above example, to access the *fname* field you would use:

```
Employee.fname = "Susan"
```

Using With

To access multiple fields at a time, you can use the *with-End with* block. The following snippet shows how to use the *with* block with the above example.

```
With Employee
    .fname = "Susan"
    .lname = "Jones"
    .empid = 1001
    .dept = 24
End With
```

The compiler will automatically bind the variable *Employee* to the individual elements within the *with* block. Not only does this mean that you don't have as much boilerplate structure is optimized and is a bit faster than using the full dot notation.

Passing Types to Subroutines and Functions

One advantage to using types in your program is that you can pass the : subroutine or function and operate on the structure as a whole. The follo shows a partial subroutine definition.

```
Sub UpdateEmployeeDept(ByRef Emp As EmployeeType)
    .
    .
    .
End Sub
```

Notice that the parameter is qualified with `ByRef`. This is important since the type within the subroutine. There are two parameter passing modes and `ByVal`.

ByRef and ByVal: A Quick Introduction

`ByRef` and `ByVal` tell the compiler how to pass a reference to the subrou you use `ByRef`, or *By Reference*, you are passing a pointer reference to any changes you make to the parameter inside the sub or func will be re variable that was passed. In other words, the `ByRef` parameter points to memory.

`ByVal`, or *By Value*, on the other hand makes a copy of the parameter, all make inside the sub or func are local and will not be reflected in the act passed. The `ByVal` parameter points to a copy of the variable not the act

The default for FreeBASIC .17 is to pass parameters using `ByVal`. In ord passed parameter, you need to specify the `ByRef` qualifier. In this examp updates the the department id of the employee type, so the parameter is that the subroutine can update the dept field of the type variable.

On the other hand you may not need to update the type as in the followi

```
Sub PrintEmployeeRecord(Emp As EmployeeType)
    .
    .
    .
End Sub
```

In this sub you are just printing the employee record to the screen or a p need to change anything in the type variable. Here the default `Byval` is u copy of the employee record to the sub rather than a reference to the va in this case, you won't accidentally change something in the type variabl intend to change.

You should only use `Byref` if you intend to change the parameter data. It `Byval` in cases where you need to have the parameter data, but want to changes to the data. These accidental changes generate hard-to-find bu

Types Within Types

In addition to the intrinsic data types, type fields can also be based on a would you want to do this? One reason is data abstraction. The more ge structures, the more you can reuse the code in other parts of your progr you have to write, the less chance of errors finding their way into your pr

Using the *Employee* example, suppose for a moment that you needed to information than just the department id. You might need to keep track of manager, the location of the department, such as the floor or the building telephone number of the department. By putting this information into a s definition, you could use this information by itself, or as part of another ty the *Employee* type. By generalizing your data structures, your program \ much more robust.

Using a type within a type is the same as using one of the intrinsic data code snippets illustrates an expanded department type and an updated

```
Type DepartmentType
```

```

    id As Integer
    managerid As Integer
    floor As Integer
End Type

Type EmployeeType
    fname As String * 10
    lname As String * 10
    empid As Integer
    dept As DepartmentType
End Type

Dim Employee As EmployeeType

```

Notice that in the `Employee` definition the `dept` field is defined as `DepartmentType` as one of the intrinsic data types. To access the department information type, you use the compound dot notation to access the `dept` fields.

```

Employee.dept.id = 24
Employee.dept.managerid = 1012
Employee.dept.floor = 13

```

The top level of the type definition is `Employee`, so that reference comes from the top level. Now a type definition as well, you need to use the `dept` identifier to access the department information within the `DepartmentType`. `Employee` refers to the `employee` type, `dept` refers to the `DepartmentType` type and `id`, `managerid` and `floor` are fields within the `department` type.

You can even carry this further, by including a type within a type within a type. You can simply use the dot notation of the additional type level as needed. While the levels of nested type definitions, it gets to be a bit unwieldy when using nested types.

With and Nested Types

You can also use the `with-End` block with nested types, by nesting the type definitions within a `with-End` block.

illustrated in the following code snippet.

```
With Employee
    .fname = "Susan"
    .lname = "Jones"
    .empid = 1001
    With .dept
        .id = 24
        .managerid = 1012
        .floor = 13
    End With
End With
```

Notice that the second `with` uses the dot notation, `.dept`, to specify the `dept` definitions. When using nested `with` blocks, be sure that you match all the statements with their correct `with` statements to avoid a compile error.

Type Assignments

Extending the idea of data abstraction further, it would be nice to be able to perform the initialization of the department type from the initialization of the employee type. In the two functions, you can easily add additional department information where you can use type assignments.

Just as you can assign one intrinsic data type to another, you can assign one type variable to another, providing they share the same type definition.

The following code snippet abstracts the department initialization function result to the department type within the `Employee` type.

```
'This function will init the dept type and return it
Function InitDept(deptid As Integer) As DepartmentType
    Dim tmpDpt As DepartmentType

    Select Case deptid
```

```

    Case 24 'dept 24
    With tmpDpt
        .id = deptid
        .managerid = 1012
        .floor = 13
    End With
    Case 48 'dept 48
    With tmpDpt
        .id = deptid
        .managerid = 1024
        .floor = 12
    End With
    Case Else 'In case a bad department id was p
    With tmpDpt
        .id = 0
        .managerid = 0
        .floor = 0
    End With
End Select

'Return the dept info
Return tmpDpt
End Function

'Create an instance of the type
Dim Employee As EmployeeType

'Initialize the Employee type
With Employee
    .fname = "Susan"
    .lname = "Jones"
    .empid = 1001
    .dept = InitDept(24) 'get dept info
End With

```

As you can see in the snippet, the dept field of the *employee* type is initialized. The *InitDept* function returns a *DepartmentType* and the compiler will

the dept field of the *Employee* record.

By just adding a simple function to the program, you have made the program maintain. If a new department is created, you can simply update the *Ini* the new department information, recompile and the program is ready to

Bit Fields

There is yet another data type that can be used in type definitions, the bit field defined as *variable_name: bits As DataType*. The variable name must be followed by a colon, the number of bits, followed by the data type. Only integer types (excluding the two floating-point types 'single' and 'double' and excluding floating-point types) are allowed within a bit field. Bit fields are useful when you need to keep information. A bit can be either 0 or 1, which may represent Yes or No, C Black and White.

The following code snippet illustrates a bit field definition.

```
Type BitType
    b1: 1 As Integer
    b2: 4 As Integer
End Type
```

b1 is defined as a single bit, and *b2* is defined as four bits. You initialize the variable by passing the individual bits to the type fields.

```
myBitType.b1 = 1
myBitType.b2 = 1101
```

The data type of the bit field determines how many bits you can declare. If the data type integer is 32 bits long, you could declare up to 32 bits in the field. However, you would declare a single bit for each field, and use a number of fields to mask that you wish to use. Using a single bit simplifies the coding you

determine if a bit is set or cleared and allows you to easily identify what type definition.

The Field Property

When you create a variable of a type definition, the type is padded in memory to allow for faster access of the type members since the type fields are aligned to a Word boundary. However, this can cause problems when trying to read a file that is not padded. You can use the `field` property to change the definition.

The `field` keyword is used right after the type name and can have the values 1 (no padding), 2 for 2 byte alignment and 4 for 4 byte alignment. If you use `field` with no padding you would use the following syntax.

```
Type myType Field = 1
    v1 As Integer
    v2 As Byte
End Type
```

For 2 byte alignment you would use `field = 2`. If no `field = property` is used, padding will be 4 bytes. If you are reading a type definition created by Fortran with default alignment, then you do not need to use the `field` property.

Quick Basic

Type Initialization

You can initialize a type definition when you dimension the type just as you do with intrinsic variables. The following code snippet illustrates the syntax.

```
Type aType
    a As Integer
    b As Byte
```

```
        c As String * 10
End Type

Dim myType As aType => (12345, 12, "Hello")
```

In the `Dim` statement, the arrow operator `=>` is used to tell the compiler the type variable. The type element values must be enclosed in parentheses by commas. The order of the value list corresponds to the order of the type elements. In this example, `a` will be set to 12345, `b` to 12 and `c` to "Hello".

You cannot initialize a dynamic string within a type definition using this method. The string must

Initializing a type definition in a `Dim` statement is useful when you need to set values for a type, or values that will not change during program execution. If values are known at compile time, the compiler doesn't have to spend cycles during runtime.

Unions

Unions look similar to Types in their definition.

```
Union aUnion
    b As Byte
    s As Short
    i As Integer
End Union
```

If this were a `Type`, you could access each field within the definition. For a `Union`, you can only access one field at any given time; all the fields within a `Union` share the same memory segment, and the size of the `Union` is the size of the largest member.

In this case, the `Union` would occupy four bytes, the size of an `Integer`, v

occupying 1 byte, the *s* field occupying 2 bytes, and the *i* occupying the field starts at the first byte, so the *s* field would include the *b* field, and th include both the *b* and *s* fields.

Types in Unions

A good example of using a type definition in a union is the *Large_Integer* in `winnt.bi`. The *Large_Integer* data type is used in a number of Windows Runtime Library. The following code snippet shows the *Large_Integer* de

```
Union LARGE_INTEGER
  Type
    LowPart As DWORD
    HighPart As Long
  End Type
  QuadPart As LONGLONG
End Union
```

The *Dword* data type is defined in `windows.h` as a FreeBASIC `UInteger`, a is defined as a *Longint*. A *Long* is just an alias for the integer data type. type occupies contiguous memory locations, so the *HighPart* field follow field in memory. Since this is a union, the type occupies the same memc *QuadPart* field.

When you set *QuardPart* to a large integer value, you are also setting the fields, which you can then extract as the *LowPart* and *HighPart*. You can i that is by setting the *LowPart* and *HighPart* of the type, you are setting th *QuadPart* field.

As you can see, using a type within a union is an easy way to set or retr of a component data type without resorting to a lot of conversion code. 7 memory segments does the conversion for you, providing that the memc sense within the context of the component type.

In the *Large_Integer* case, the *LowPart* and *HighPart* have been defined appropriate component values. Using values other than *Dword* and *Long* \

correct values for *LowPart* and *HighPart*. You need to make sure when doing a union, you are segmenting the union memory segment correctly within the

Unions in Types

A union within a type definition is an efficient way to manage data when you can only have one of several values. The most common example of this is the `Variant` type found in other programming languages.

FreeBASIC does not have a native Variant data type at this time. However, by using the `Extend` module, you could create a Variant data type for use in your program.

When using a union within a type it is common practice to create an *id* field that indicates what the union contains at any given moment. The following code illustrates this concept.

```
'Union field ids
#define vInteger 0
#define vDouble 1

'Define type def with variable data fields
Type vType
    vt_id As Integer
    Union
        d As Double
        i As Integer
    End Union
End Type
```

The union definition here is called an anonymous union since it isn't defined. The `vt_id` field of the type definition indicates the value of the union. To initialize the union, you would use code like the following.

```
Dim myVarianti As vType
Dim myVariantd As vType

myVarianti.vt_id = vInteger
myVarianti.i = 300

myVariantd.vt_id = vDouble
myVariantd.d = 356.56
```

myVarianti contains an integer value so the *id* is set to *vInteger*. *myVariantd* contains a double so the *id* is set to *vDouble*. If you were to create a subroutine that takes a variant as a parameter, you could examine the *vt_type* field to determine whether an integer or double had been passed to the subroutine.

You cannot use dynamic strings within a union.

Using a combination of *unions* and *types* within a program allows you to create data types that have a lot of flexibility, but care must be taken to ensure that they are used correctly. Improper use of these data types can lead to hard-to-debug errors. However, when used correctly, the benefits outweigh the risks and once mastered, are a powerful tool.

New to Programming?



If you're new to programming in general, you should probably learn what

How Your Program Is Run

- What a Compiler Is
- Syntax
- Program Flow

Variables

- Basic DataTypes

Input/Output (IO)

The above being the most important programming concepts for an absolute beginner in FreeBASIC. It is also important to learn how to use the manual, locate and have manuals with both descriptions and demonstrations. ALWAYS refer to what you want is in the manual, and if it's not, it can be added.

This tutorial's on Version 1.0. Don't care for the revision number ^^;;

How Your Program is Run

What a Compiler Is

FreeBASIC is a compiled programming language, rather than interpreted. It handles "PRINT" or "SLEEP" and translates that directly into Assembly or Machine Code. In general, you will never code in Machine Code, no matter how "low level" it is.

FreeBASIC is a High Level programming language. FreeBASIC makes it easier to do programming, you don't have to worry about the more complex areas of lower level, the programmer has the advantage of manipulating the compiler to know more about internals.

Your compiler of choice will depend on your situation. If you want complete control over the code in ASM or C. However, as computers and compilers have progressed, the details of your code. In many ways, the entire purpose of higher level programming is to let FreeBASIC handle many optimizations and improvements that you would otherwise have to do yourself.

level areas of control if you wish. One problem with this, however, which levels of implicit actions being taken by the compiler. If you want to work explicitly control certain aspects of your code.

Syntax

Syntax is how words and commands are grouped together in programm consistent, will lay down rules as to how you will structure your program.

For example, in programming, you will come across the task of calling c tell you how you can call this command, and what is or isn't allowed. The that could occur in a more "syntax free" (what is essentially impossible in

The syntax for FreeBASIC generally goes as follows: CommandName [/

While the above may look confusing at first, it's actually very simple. All arguments after the command. The comma is what separates the argun can assume that Draw will draw something, Circle will be the shape that syntax rules for that command may look something like this: Draw [Shap

FreeBASIC is **not** case sensitive. Calling a command 'DRaW' is the sa

Program Flow

FreeBASIC's code is read from the TOP of the code, to the BOTTOM, o code for that line is read by the computer, the command that's on the lin code tells it to). Example code can be:

```
Print "HI"  
Sleep
```

Since the code PRINT is on the line above SLEEP, PRINT will be run fir executing.

Comments can be made in FreeBASIC, which are ignored, and will not l them with ', or can be multiple line comments if you begin them with '/' ar

In our program, we created foo. FOO was created as an INTEGER (A data type). We used the command PRINT, which PRINTS information on our screen. We PRINTED a message which pauses our program until we hit a key.

Basic DataTypes

Variables are a tough subject, I think, to begin with in programming. The question is 'The kind of data that's held in this type of variable', and you wondered what these types of variables:

Integer - Hold numbers WITHOUT DECIMAL PLACES. Will generally be smaller than a Double (the variable it's actual value will be)

String - A nice feature in FreeBASIC. STRING is a datatype which holds text and cool information to put on the screen, such as cooking directions.

Remember, follow the proper DIM syntax. DIM variablename as INTEGER in our program. You can replace INTEGER with DOUBLE, or STRING. BE CAREFUL! Do not give a STRING the value of 5! You can however, give it the value of 5. A double equal "5", as "5" is a string, and not a number.

Here is a really cool example, which demonstrates how you can use variables.

```
' Create the variable MyName. Assign it's value to Alex
Dim As String MyName = "Alex"

' Print The MyName variable
Print MyName

' pause the program until the user hits a key.
Sleep
```

Input/Output

Input is the receiving of information. When you get input on something, (GETTING SOMETHING, Retrieving Something)

Output is the sending of information. When you output to something, you

Input and Output are often put together, and are shortened as I/O, or IO.

FreeBASIC has MANY methods of input and output. For a beginner, no knowledge of variables and more complex forms of programming. We're

You remember the command PRINT in the above examples? That's OUTPUT of output, and it's easy to learn, too! You just call the command PRINT, enclosed in Double-Quotes. If you want to print variables, you just give F

Print [WhatToPrint]

Example:

```
' Print the words, HI! to the screen
Print "HI!"

' create a new integer and name it foo. Give it the value 10
Dim As Integer foo = 10

' Print the value of foo.
Print foo
Sleep
```

INPUT isn't much harder, either. However, whenever you input, you have output, we have to give the PRINT command something to output. We are the user inputs.

- 1) We need a variable to store that information in.
- 2) We need to call a command to get input.
- 3) We need to print the input to make sure we stored the information correctly.

I know how to do 1 and 3, but what about 2? We're going to learn a new one we will use the command, INPUT.

Input's syntax is as follows: INPUT [VariableToInputTo]

You can also use input like so: INPUT [Output String To Tell User What to

The first version of INPUT will let you get input, and put it right into the variable asking for input. This way, the user will know what to input! Alternatively, you can use a message, but sometimes being able to put related code on one line is a

Example:

```
' Create a string. We will hold the user's name in it
Dim As String MyName

' Get the user's name!
' The message Please Enter Your Name is posted on the screen
' and then the user has a chance to enter in their name
Input "Please enter your name!", MyName

' Print the user's name that we just got.
' Just like input, we can print several messages on the screen
' execute different types of commands by separating them with a colon
Print "Your Name Is: ", MyName

' pause the program until the user hits a key
Sleep
```

That demonstrates both INPUT and OUTPUT! Both are essential in programming things, as well as INPUT. You might be getting input from a robotic arm's power drill rather than a monitor. It really depends on the hardware and what

At the time, and in most cases, you don't have to worry so much about variables and standard I/O functions. More advanced methods of I/O let you decide what

output to).

Programming Definitions

Argument: See Parameter

ASM: The lowest level code that a human will want to read. This can be

Compiling: The process of turning text in one language to another. Ex: B code.

Machine Code: 0's and 1's. This is *the* code that your computer will un

Parameter: Data that you pass to a command you call in programming. I do something, or what they will do. Passing a parameter 'Rectangle' to a screen.

Pixel: One 'dot' on your monitor. Monitors are made up of thousands of t the pixel variable that the monitor receives. Believe it or not, even your h

Syntax: How words are grouped together. Your syntax in programming a that only logical code is allowed. Ex: Print "Hi". PRINT is the COMMAND

Variable: A word that holds data in programming. You assign these word

Compiling a Big QB program in FB



Let's try to compile a big (4000+ lines) graphical QB program in FreeBasic, to see how compatible FB is with QB.

As an example I will use Jark's TCRay a great raytracer with quadric, cubic and quadratic shapes, perlin noise programmed in 2004. You can get TCRay.zip from

<http://www.mandelbrot-dazibao.com/Programs/Programs.htm>

Notice TCRay is a QB4.5 interpreted program, Jark never had the patience to compile his work, he just tested it interpreted and went on adding features.

The program is made of has 3 files:

TcRay21C.bas - The Main file.

TcLib17L.bas - The SVGA graphics library.

Tclib17.bi - The include file for the library.

Porting TCLib17.bas

In TCLib17.bas

It is a "pure QB" SVGA library. Most of its functions are obsoleted by FB as they are implemented as QB-style keywords. I had my share in developing that lib so you can trust me for this part ;)

Comment out the contents of the ClearScreen sub and add this
CLS

Comment out the contents of the Point24 sub and add this:

```
a& = Point(x%,y%)
red% = a& Shr 16
green% = (a& Shr 8) And 255
blue% = a& And 255
```

Comment out the contents of the Pset24 sub and add this:

```
PSet (x%,y%), red% Shl 16 Or green Shl 8 Or blue
```

Comment out the contents of the Screenshot sub and add this:
BSave Name\$+" .bmp"

Comment out the contents of the SelectVga sub, we will work with a fixed size most pc's will support. Comment out the contents of the SetText sub we are able to output text in HiRes graphics so mode switching is not required.

Comment out the contents of the SetVGA Sub excluding the Powers of two calculation at the end and add these four lines:

```
Screen 20,32 '1024x768, 32 bits  
scrheight=768  
scrwidth=1024  
Fullscreen
```

In TCRay17.bas

Add a SetSVGA as the first line in the Menu sub (we are not switching modes so mode must be set before outputting text),

Compiling

Ok, stop trusting me, now you can start trying to compile. You'll receive some errors.

Compile with: *fbcc -s gui -w 1 -lang qb TcRay21C.bas TcLib17L.bas*

I warn you all changes required except two come from a couple of (wise) limitations in the FB syntax:

- A variable name can't be a keyword plus a type suffix
- A simple variable can't have the same name as an array

In TCLib17.bi

ERROR: Duplicated definition, found 'RGB' (**RGB** is a keyword in FB)

Add:

```
#undefine RGB
```

Before the line giving the error.

ERROR: Duplicated definition, found 'ScreenRes' (**ScreenRes** is a keyword in FB)

Add:

```
#undefine ScreenRes
```

Before the line giving the error.

ERROR: Duplicated definition, found 'Name' (**Name** is a keyword in QB)

Add

```
#undefine Name
```

Before the line giving the error.

In TCRay17.bas

ERROR: Duplicated definition, found 'Acos' (**Acos** is a keyword in QB)

Add

```
#undefine Acos
```

Before the line giving the error.

ERROR: Argument count mismatch Clear (**Clear** is not required in FB, the keyword has been reused (not a clever decision?))

'comment out CLEAR

ERROR: Illegal specification, at parameter 2 (Type) of Init.Cubic() (**Type** is a keyword in QB)

We can undefine it so search and replace type\$ to _type\$

ERROR: Expected 'END IF', found 'END' END FUNCTION

This is an error caused by a quirk introduced in FB. Single line **if**'s having a colon after THEN require an ENDIF, it has to do with macros... What reason had Jark to put colons past his THEN's escapes me. QB does not require them at all and FB behaves as expected without them. Remove all colons after THEN keywords. Search and Replace *THEN :* to *THEN*

ERROR: Array access, index expected, before '=' $x_n = x * x - y * y + z * z$

We have an array names x_n , and a variable named x_n . Substitute x_n with $_x_n$ in the lines which error when you try to compile.

The same error with x_0 , we have an array called x_0 .

Substitute x_0 with $_x_0$ in the lines which error when you try to compile.

ERROR: Array access, index expected, before '*' $dAdR = Amplitude * dAdR * drdY$. Same problem with Amplitude.

Substitute Amplitude with `_Amplitude` in the lines which error when you try to compile.

Ok. At this point all modules compile. We're now going to fix a few linker errors.

Linker Errors

After compiling the linker ties together all the modules with a runtime library, finds the final addresses of every sub/function and substitutes the labels in the calls with these addresses. If a sub/function is called in the code and its nowhere to be found, the linker complains and gives us the name of the offending function. It can't give us the line numbers (the linker doesn't work with the source) so we will have to do a text search to find where the problem occurs. Notice the linker gives us "mangled" function names (an ampersand and the size of the parameters passed is added to the end), just ignore the ampersand and what's after.

TcRay21C.o:fake:(.text+0x174d): undefined reference to `LINE24@20'

A call to an undefined Line24 sub is made in the program, you can find this call inside Draw.Axis, in TCRay21.bas, a sub that's it's never called (you can do a search to confirm it)

Probably the QB4.5 compiler would complain too about this. (Remember this program never compiled in QB4.5) Just comment out the contents of the sub Draw.Axis

TcRay21C.o:fake:(.text+0x181b3): undefined reference to `FFIX@0'

Ffix was that useful v1ctor's floating point patch for QB 4.5. It's not needed in FreeBASIC. Just comment out the line calling it just after the declarations in tcray21c.bas

And that's all, the program compiles and works. Not a lot of changes for 4000+ lines...

Enjoy!

antonigual [at] eic [dot] ictnet [dot] es

Introduction by Lachie Dazdarian

The objective of this series of lessons is to help newbies who know very little programming in FreeBASIC necessary to create any computer game. So I'll use the word (well, it's an acronym) "BASIC" and not FreeBASIC, because if you know BASIC or any other variant of BASIC, these lessons should be easy to understand.

I'm starting this series because I feel that tutorials of this kind were always lacking, even before FreeBASIC. I've corresponded during my programming with newbies, and they all had almost identical problems when trying to program. Beginners need quite well and on what way the stuff needs to be explained and the problems I had with using separated routines that were never merged into one game. The breaking point for me was the moment when I discovered R.E.Lope (R.E.Lope) and the scrolling engine that was created with it. That scrolling engine mechanics and expand on it (with some help from R.E.Lope). In one sentence, most of the stuff (necessary to complete a game) by myself. It's like driving a car: actual skill lasts for one second.

So that's my goal with this series. To learn you enough so you would be able to learn new things is to see them applied. Many tutorials fail in this help from more expert programmers, but the point is that you don't need to know depends on the type of game you are developing and the graphics library.

The example programs and mini-games we'll create will be coded in GFA (GFA library). Lynn's Legacy, Arkade, Mighty Line and Poxie were coded in it. These games are good references. But don't worry. Switching from one graphics library you know how to code in at least one.

This tutorial will not deal with raycasting engines (3D programming) or scrolling engines, but if you are a beginner, you NEED the following lessons FIRST.

Since we are going to code in FreeBASIC you need to get FreeBASIC first. <http://www.freebasic.net> (the examples were compiled with version 0.18) I recommend FBIDE or FBEdit.

Example #1: A simple program - The circle moves!

We'll start with some elementary stuff. The first program we'll code will not use graphics from external files (usually BMP images) is always a dirty business on this. Be patient.

The program we'll create will allow you to move a circle around the screen, making it we'll learn important facts and a lot of elementary statements as well with GFXlib.

As we are using GFXlib you need to be aware of the gfxlib.txt file (GFXlib.txt) in the /FreeBASIC/docs directory. That's our Bible and very useful with these kinds of every statement used in the example programs (most likely). This document has moved on with new versions, so be sure to refer to this online FreeBASIC documentation.

Open a new program in FBIDE. First thing we'll do is set the graphic mode. We'll set the program's graphic resolution and color depth in bits (8-bit, 16-bit, ...) to a standard 256 colors mode (8 bits per pixel). The graphic mode is set with

```
Screen 13,8,2,0
```

13 means 320*200 graphic resolution, 8 means 8-bit graphics, 2 means 256 colors (input 1 for full screen mode). Minimum of 2 work pages is recommended. These things will become clearer a little bit later. For more details about the SC mode, see the documentation or FreeBASIC Wiki (a more "advanced" version of the SC mode).

The next thing we'll do is set a loop that plays until the user pushes the Enter key. This is not a good way of any program, not just a computer game. Coding a program on a way that forces the user to type something in is a BAD and WRONG way to program. We'll use loops as places where the program waits for the user to do something and where the program executes some routine according to user's actions. Things that are not controlled by the player (enemies) are managed/moved. Loops are a

If you are aware of all these things, you can skip to the end of this section (comments). If there is something in it you don't understand, then get back

We can set a loop on more ways (with WHILE:WEND statements, using way is to use DO...LOOP. This type of loop simply repeats a block of sta the condition(s) after LOOP with UNTIL. Check the following code:

```
Screen 13,8,2,0 ' Sets the graphic mode
Do
' We'll put our statements here later
Loop Until Inkey$ = "Q" Or Inkey$ = "q"
```

If you compile this code and run it, you'll get a small black empty 320*20 the letter Q (you might need to hold it). The program simply loops until y lower case "Q" symbol in case Caps Lock is turned on on your keyboard key pushed on the keyboard. I will explain later why it shouldn't be used

To draw a circle we'll use the CIRCLE statement (refer to GFXlib's docu

```
Screen 13,8,2,0 ' Sets the graphic mode
Do
Circle (150, 90), 10, 15
Loop Until Inkey$ = "Q" Or Inkey$ = "q"
```

The last code draws a small circle on coordinates 150, 90 with a radius (which you can check if you compile the code. So how to move that circle VARIABLES. For this we'll use two variables named circlex and circley. (

```
Dim Shared As Single circlex, circley
Screen 13,8,2,0 ' Sets the graphic mode
circlex = 150 ' Initial circle position
```

```
circley = 90

Do

Circle (circlex, circley), 10, 15

Loop Until Inkey$ = "Q" Or Inkey$ = "q"
```

This makes no change in the result of our program, but it's a step to what amounts to which circlex and circley equal to change the circle's initial position to move the circle we need to connect circlex and circley variables

We declared first two variables in our program. Since FreeBASIC ver.0.15 can be declared, although if you use -lang qb command line during compiling compatibility dialect (I don't recommend it as it will keep you deprived of default FB compatibility already provides and will provide). For more info see FreeBASIC wiki - Using the command line. Variables are declared (dime

```
Dim variable_name [As type_of_variable]
```

Or...

```
Dim [As type_of_variable] variable1, variable2,
```

The data inside [] is optional and the brackets are not used. Types of variables are SHORT, INTEGER, STRING, SINGLE, DOUBLE and few others, but I'd like to mention that at a low level. What you need to know now is that you should declare variables carefully when they represent graphics data (memory buffers holding graphics) or when they represent numerical data (number of lives, points, etc.). Variables that need decimal precision are usually variables used in games which rely on physics formulae like arc length (gravity effect). Simply, the difference between the speed of two pixels per second is most often too large, and in those limits you can't emulate effects like

Also, behind DIM you should put SHARED which makes that the specific subroutines). Don't use SHARED only with variables declared inside subroutines. To declare ARRAYS inside a subroutine, I advise you to replace DIM with Like YourName = "Dodo", but you need to declare YourName AS STRING

Now I will introduce a new statement instead of INKEY\$ which can detect a keypress responsive (perfect response) than INKEY\$. The flaw of INKEY\$, as we probably were able to detect when trying to shut down the previously code, is that it can detect one keypress at any given moment which renders it completely unusable.

The substitute we'll use is MULTIKEY (a GFXlib statement) which features the scancode of the key you want to query. You might be lost now. DOS scancode stands for a certain keyboard key. If you check Appendix A of the GFXlib code stands for. For example, MULTIKEY(&h1C;) queries if you pushed the key with scancode &h1C. Constants like it's explained in Appendix A of the GFXlib (fbgfx.bi) into your source. What's a .bi file? Well, it can be any kind of module file and which can feature various subroutines (if you don't know what a subroutines are, see the declarations used in your main module. The code you need to add are the following:

```
#include "fbgfx.bi"  
Using FB
```

It's best to put these two lines somewhere on the beginning of your program. You don't need to set a path to fbgfx.bi since it's placed in the /FreeBASIC/include/ directory if it's not in that directory or not in the directory where the source code is. This way you can be accessing GFXlib symbols without namespace, meaning, without having to use the symbol. Refer to FreeBASIC Wiki on USING.

Now the fun starts.

We will add a new variable named circlespeed which flags (sets) how many times the program will loop (loop). The movement will be done with the arrows key. Every time the user pushes a key, the program will change either circlex or circley (depends on the pushed key) using the following code:

```

#include "fbgfx.bi"
Using FB

Dim Shared As Single circlex, circley, circlespeed

Screen 13,8,2,0 ' Sets the graphic mode

circlex = 150 ' Initial circle position
circley = 90
circlespeed = 1 ' Circle's speed => 1 pixel per

Do

Circle (circlex, circley), 10, 15

' According to pushed key we change the circle's
If MultiKey(SC_RIGHT) Then circlex = circlex + circlespeed
If MultiKey(SC_LEFT) Then circlex = circlex - circlespeed
If MultiKey(SC_DOWN) Then circley = circley + circlespeed
If MultiKey(SC_UP) Then circley = circley - circlespeed

Loop Until MultiKey(SC_Q) Or MultiKey(SC_ESCAPE)

```

As you see we also changed the condition after UNTIL since we are using the program by pressing ESCAPE too (I added one more condition).

If you compile the last version of the code, two things we don't want to happen: if you move the mouse too fast you won't even notice the movement of the circle, and the circle will be drawn at different coordinates in previous cycles will remain on the screen). To avoid this, we added a statement (clears the screen) in the loop so that in every new cycle the circle is drawn before the new one is drawn.

To reduce the speed of the program the quickest fix is the SLEEP command. We can specify the amount of time that has elapsed (in milliseconds) or a key is pressed. To escape the loop we can use milliseconds, 1. This statement is also an efficient solution for the 100% CPU use that statement any kind of FreeBASIC program with a loop (even the

cycles and make all the other Windows tasks you might be running to co
with other tasks while that kind of FreeBASIC program is running. Err...th
programmers that have released FreeBASIC games so far did not bothe

Copy and paste the following code and compile it:

```
#include "fbgfx.bi"
Using FB

Dim Shared As Single circlex, circley, circlespe

Screen 13,8,2,0 ' Sets the graphic mode

circlex = 150    ' Initial circle position
circley = 90
circlespeed = 1 ' Circle's speed => 1 pixel per

Do

Cls
Circle (circlex, circley), 10, 15

' According to pushed key we change the circle's
If MultiKey(SC_RIGHT) Then circlex = circlex + c
If MultiKey(SC_LEFT) Then circlex = circlex - ci
If MultiKey(SC_DOWN) Then circley = circley + ci
If MultiKey(SC_UP) Then circley = circley - circ

Sleep 10, 1

Loop Until MultiKey(SC_Q) Or MultiKey(SC_ESCAPE)
```

Viola! Our circle is moving and "slow enough".

The last version of the code does not represent the desirable way of coc
make this lesson easy to understand. What we need to do next is declar

declared in any "serious" program, and show why we are having two wo

The way variables are declared in the above code is not the most conve amount of variables usually associated to several objects (an object can defined with MORE THAN ONE variable).

So first we'll define a user defined data type with the statement TYPE th with me). We'll name this user data type ObjectType. The code:

```
Type ObjectType
    x As Single
    y As Single
    speed As Single
End Type
```

After this we declare our circle as an object:

```
Dim Shared CircleM As ObjectType
' We can't declare this variable with "Circle"
' since then FB can't differ it from
' the statement CIRCLE, thus "CircleM".
```

How is this method beneficial? It allows us to manage the program varia Instead of (in this example) having to declare each circle's characteristic simply use a type:def that includes all these variables and associate a v: CircleM). So now the circle's x position is flagged with CircleM.X, circle's with CircleM.speed. I hope you see now why this is better. One user def variables or arrays. In this example you can add another object with son ObjectType which would allow us to manage 8 "evil" circles with a specifi the variables from the ObjectType type:def (x, y, speed), and these circle way. In the next lesson all this will become more clear. Have in mind tha a type:def. This is only for "objects" in your game that are defined (chara determined by health, money, score, strength, etc.).

After the change the final version of the code looks like this:

```
#include "fbgfx.bi"
Using FB

' Our user defined type.
Type ObjectType
  x As Single
  y As Single
  speed As Single
End Type

Dim Shared CircleM As ObjectType
' We can't declare this variable with "Circle"
' since then FB can't differ it from
' the statement CIRCLE, thus "CircleM".

Screen 13,8,2,0 ' Sets the graphic mode
SetMouse 0,0,0 ' Hides the mouse cursor

CircleM.x = 150 ' Initial circle's position
CircleM.y = 90
CircleM.speed = 1 ' Circle's speed => 1 pixel per

Do

Cls
Circle (CircleM.x, CircleM.y), 10, 15

' According to pushed key we change the circle's
If MultiKey(SC_RIGHT) Then CircleM.x = CircleM.x + CircleM.speed
If MultiKey(SC_LEFT) Then CircleM.x = CircleM.x - CircleM.speed
If MultiKey(SC_DOWN) Then CircleM.y = CircleM.y + CircleM.speed
If MultiKey(SC_UP) Then CircleM.y = CircleM.y - CircleM.speed

Sleep 10, 1 ' Wait for 10 milliseconds.

Loop Until MultiKey(SC_Q) Or MultiKey(SC_ESCAPE)
```

You will notice I added one more statement in the code. The SETMOUSE cursor (first two parameters) and shows or hides it (third parameter; 0 hides these parameters in every program AFTER the SCREEN statement (IMMEDIATE program is going to feature a mouse controllable interface, you will most likely be using that line way too often).

Download the completed example with extra comments inside the source code.

Phew, we are done with the first example. Some of you might think I went a little far but I think a little dance was needed to make the next examples and lessons a more enjoyable experience.

Nevertheless, this example is far from what we want, right? So the next example will be more complex than the first from external files among other things.

Example 2: A warrior running around a green field

In the next example we will be applying all the knowledge from the first example and we will go into every statement again. I will explain every new statement and just like the first example.

In this section we'll start to code our mini-game which won't be complete yet but it will be a program where a warrior runs around a green field (single screen).

First I'll show you what graphics we'll be using. We are going to work in 256 colors mode (256 colors mode). For the warrior character from my first game Dark Quest.

<http://hmcsoft.org/fb/htpagl1-sprites.png>

As you see this image features 12 sprites of our warrior, each 20*20 pixels (12 sprites for the animation) and one sprite for each direction when the warrior is swinging a sword. The warrior is implemented in the first lesson but will become necessary later.

Second image is the background image which you can check/download (it's a BMP image).

Download both images and place them where you will place the source code.

the end of this section.

On the beginning of our program we should include `fbgfx.bi`, same as in graphic mode. The code:

```
#include "fbgfx.bi"
Using FB

Screen 13,8,2,0 ' Sets the graphic mode
SetMouse 0,0,0 ' Hides the mouse cursor
```

Now we will declare two memory pointers that will point to memory buffers (the sprites and one for the background).

The first pointer we'll name `background1` and declare it with the following:

```
Dim Shared background1 As Any Ptr
```

ANY PTR tells us that `background1` will actually be a memory pointer. A compiler checking for the type of data it points to. It is useful as it can point to memory buffers because we will allocate memory for our graphics using the `IMAGECREATE` function. `IMAGECREATE` allocates the right amount of memory for a piece of graphics (sprite/image) and we would have to do it manually, meaning, calculate the needed amount of memory based on the width and height of the image. `IMAGECREATE` does this for us. As `IMAGECREATE` returns a pointer to it and not a variable. Don't worry if you don't know anything about pointers (I will explain and comprehend this tutorial).

The next pointer we'll declare will point to the memory buffer that holds the sprites. We'll declare it as a single dimension array, each element in the array represents a sprite.

```
Dim Shared warriorSprite(12) As Any Ptr
```

Both these lines should be put in the code before the SCREEN statement, Subroutine declarations, then variable declarations, then extra subroutine code. The beginning of our program should now look like this:

```
#include "fbgfx.bi"
Using FB

Dim Shared background1 As Any Ptr ' A pointer to a buffer holding the background
Dim Shared WarriorSprite(12) As Any Ptr ' A pointer to an array of buffers holding the warrior sprites

Screen 13,8,2,0 ' Sets the graphic mode
SetMouse 0,0,0 ' Hides the mouse cursor
```

After the screen resolution, color depth and number of work pages are set, we will load the graphics onto it since we don't want for the user to see all of the program at once. To accomplish that we'll use the SCREENSET statement. The first parameter is the work page (first parameter) and the visible page (second parameter). In our case we will use work page 0 and visible page 1. After using 'SCREENSET 1, 0' every time we draw or load a graphic, the graphic will be loaded/drawn on the work page and won't be visible to the user until we use SCREENSET with different parameters (SCREENSET 1, 1). This allows us to load graphics that we want for the user to see and delete it before copying the content on the work page. Flipping is also useful in loops with "graphics demanding" programs to avoid flickering.

So Biff wants to have a high score table in his game



Written by Lachie Dazdarian (September, 2007)

Introduction

On more than one occasion I was inquired by a programming newbie about how to load a new high score properly, and then save the modified high scores table. Using the same set of routines for high scores since the days of Ball Blaster, flexibility (plus few fixes) there, something that was long needed to be done. The tutorial will also point you out to some useful (for high scores table routines) not written by me.

Let's do it!

It's fairly obvious we'll need two separate subroutines, one for loading/reading and one for saving. We'll start with loading/reading of a high score table, as that part is easier. The subroutine for reading a high score table should work relatively simply by reading appropriate variables and then printing them on the screen, this part being the more difficult (printing, position of the high score table, its formatting, etc.). First, we should create a text file containing our name and score entries.

```
FRED
10000
BILL
9000
SARAH
8000
BOB
7000
RED
6000
SUE
5000
DAVID
4000
```

```
GREG
3000
TIM
2000
GEORGE
1000
```

It contains 10 high score entries, formatted with name followed by the score. You can pick one where all the names are listed first, and then followed by the scores, so we'll work with the one I started with.

This file will be used with the following 'ReadHighScore' subroutine.

Let's start our main program with some needed initiation statements:

```
#include "fbgfx.bi"
Using FB

Const num_of_entries = 10
```

'num_of_entries' will flag the number of score entries (names or scores) in the 'high_score.dat' file (not lines, but high score ENTRIES!).

We should now declare our subroutine with:

```
Declare Sub ReadHighScore (highscore_file As String)
```

The 'highscore_file' variable will flag the file you want for the 'ReadHighScore' subroutine. This adds some flexibility to it.

After this, we should declare the following variables:

```
Dim Shared workpage As Integer
Dim Shared hname(num_of_entries) As String
Dim Shared hscore(num_of_entries) As String
```

'workpage' variable is not related to this tutorial and will be used to swap
'hname' array will hold the name entries, while 'hscore' array will hold the

Finally, let's initialize our screen and work/visible pages with:

```
ScreenRes 640, 480, 32, 2, GFX_ALPHA_PRIMITIVES+GFX_
ScreenSet 1, 0
```

Following this code we should place this:

```
ReadHighScore "high_scores.dat"
End

Sub ReadHighScore (highscore_file As String)
End Sub
```

You can compile this code, but nothing will happen as the 'ReadHighScore' sub is not called.
We need to start it by opening the 'high_scores.dat' file and reading the file opening in FreeBASIC if not familiar with it.

As we want to open the file using a FREE file handle, we need to dimension

```
Dim free_filehandle As Integer

free_filehandle = FreeFile
```

We should now open the high score file with:

```
Open highscore_file For Input As #free_filehandle
```

After the file is opened for reading (FOR INPUT), let's use a for loop to r

```
For count_entry As Integer = 1 To num_of_entries  
Input #free_filehandle, hname(count_entry)  
Input #free_filehandle, hscore(count_entry)  
' If the end of file is reached, exit the FOR loop.  
If EOF(free_filehandle) Then Exit For  
Next count_entry
```

Note how the 'count_entry' variable is used and how for each entry the r the name with the top score, while 'hscore(1)' the top score. 'hname(num 'hscore(num_of_entries)' the lowest score in the high score table.

Don't forget now to close the file with:

```
Close #free_filehandle
```

All we need now is a loop that will display all these names and scores, n

```
Do  
  
ScreenLock  
ScreenSet workpage, workpage Xor 1
```

```

Line (0,0)-(639,479), RGBA(0, 0, 0, 255), BF

Draw String (285, 120), "TOP SCORES", RGBA(255,255,

For count_entry As Integer = 1 To num_of_entries
Draw String (270, 140 + count_entry * 12), hname(count_entry)
Draw String (340, 140 + (count_entry) * 12), hscore(count_entry)
Next count_entry

Draw String (245, 400), "Press ESCAPE to exit", RGBA(255,255,255)

workpage Xor = 1
ScreenUnlock

Sleep 10

Loop Until MultiKey(SC_ESCAPE)

```

A simple DO...LOOP that ends when the user pushes ESCAPE.

I used Draw String to print the names and the scores. Another FOR loop prints the score under the next higher one (note how the Y position of the text to draw is 12 more than the previous one, which creates space between scores vertically). I also used a small trick to display each name and score on a new line.

After placing all this code in the 'ReadHighScore' subroutine, you can call it from the main program like this:

Now when we are done with the easy part of the problem, let's move on to the hard part.

I constructed the 'WriteHighScore' subroutine like this:

```

Sub WriteHighScore (highscore_file As String, userscore As Integer)

```

Which means it will be called with a high scores table file and a score we want to add to the table, no code will be executed.

This subroutine should start with the following code:

```
Dim free_filehandle As Integer

Dim startwrite As Integer

free_filehandle = FreeFile

Open highscore_file For Input As #free_filehandle

For count_entry As Integer = 1 To num_of_entries
Input #free_filehandle, hname(count_entry)
Input #free_filehandle, hscore(count_entry)
' If the end of file is reached, exit the FOR loop.
If EOF(free_filehandle) Then Exit For
Next count_entry

Close #free_filehandle
```

As you see it starts as the 'ReadHighScore' subroutine. In order to evaluate file containing our high score entries and store them in appropriated variable the high score table (on which position).

The code that follows should be opened with an IF clause that will execute the high score table (naturally):

```
If users_score > hscore(num_of_entries) Then

For check_score As Integer = 1 To num_of_entries

If users_score > hscore(check_score) Then
InputName
' Record the position where the new score is
' to placed and exit FOR loop.
startwrite = check_score
Exit For
```

```
End If
```

```
Next check_score
```

The FOR loop 'goes' through the high score entries from the highest to the lowest (flagged with 'startwrite' and 'check_score') where our new entry will be placed. If the user's score ends up being higher than the current top score in the high score table, the new entry needs to be placed at the top. 'InputName' is a subroutine we'll create later, and inside it

What follows is the 'nexus' of our routine, the code that places the new high score position down.

Check the following code:

```
If startwrite = num_of_entries Then
    hscore(startwrite) = users_score
    hname(startwrite) = playername
Else
    For write_pos As Integer = (num_of_entries - 1) To startwrite
        hscore(write_pos + 1) = hscore(write_pos)
        hname(write_pos + 1) = hname(write_pos)
    Next write_pos
    hscore(startwrite) = users_score
    hname(startwrite) = playername
End If
```

First condition checks if the new entry is the lowest (last) in the high score table. If not, a lower score as there are none, but only replace the lowest score entry with the new one. If this is NOT the case, a FOR loop is executed which loops from the lowest score to the highest score, meaning, from bottom to top.

For example, if our high score table has 10 entries and the new entry needs to be placed at the 9th position, values from 'hscore(9)' and 'hname(9)' are passed to 'hscore(9+1)' and 'hname(9+1)' respectively.

are passed to 'hscore(8+1)' and 'hname(8+1)'. And so on.

After the FOR loop we need to input the new entry on its appropriate position 'playername', where 'playername' will be inputted inside the 'InputName'

The last thing in the 'WriteHighScore' sub we need to do is to store the r

```
free_filehandle = FreeFile

Open highscore_file For Output As free_filehandle
For count_entry As Integer = 1 To num_of_entries
Print #free_filehandle, hname(count_entry)
Print #free_filehandle, hscore(count_entry)
Next count_entry
Close free_filehandle
```

Note how FOR OUTPUT is used and PRINT for writing data into external files. After this I placed a 'ReadHighScore' call and closed with END IF as I finished being inputted in it.

All we need now is to create the 'InputName' sub like this:

```
Sub InputName

ScreenSet workpage, workpage Xor 1
ScreenSet 0,0
Line (0,0)-(639,479), RGBA(0, 0, 0, 255), BF
Locate 12, 17
Input ; "Please input your name: ", playername

End Sub
```

Of course, this will look totally different in your game. Perhaps you'll ask the player (e.g. when he/she starts a new game). Just have in mind you need one.

To test the routines just place...

```
ReadHighScore "high_scores.dat"  
WriteHighScore "high_scores.dat", 4500  
End
```

...after first SCREENSET (outside subroutines). Change the second parameter in the high score table. I'm sure you are aware that when calling 'WriteHighScore' but with a variable in which you'll store player's score, whatever that may be.

What's next?

The only other things I wish to share regarding this issue is related to high score encryption. Using encryption are not by me, I will only brush off them and provide them in an example. Encryption is done using two functions, 'neoENCpass' and 'neoENCdep' (high score entry string in our case) and password, password being any string (of course).

Just after you retrieve an string entry from a file you decrypt it like this:

```
Input #free_filehandle, hname(count_entry)  
neoENCdep SAdd(hname(count_entry)), Len(hname(count_entry))
```

With 'hscore' variables, being INTEGER, we need to use a temporary string variable. The only annoying feature of this method is the fact you need a separate project will work only if the high score file is previously encrypted. I provide you keep a backup of your high score file in a separate folder (I also provide a routine. Instead of encryption you can use BINARY files, which I don't know how to do (the nick of time), and which also AREN'T the same as ENCRYPTION. Encryption is done using a password (well, most people), while BINARIES can be read by anyone but you need to change the encryption passwords inside it.

Anyway, you might not need or prefer encryption at all. But I personally I can change/read them with Notepad. Unencrypted high scores might kill your program. Name inputting routine I won't go describing as that's irrelevant. You have custom font printing libraries) and allows you to limit the number of characters.

to him.

Download the extended example (with encryption and better name input
http://lachie.phatcode.net/Downloads/Managing_A_High_Score_Table.z

And that's it for this tutorial.

Until next time, have fun!

A tutorial written by Lachie D. (mailto:CHR\$(58)lachie13 CHR\$(64)yah

The IF Statement



Written by *rdc*

You can think of the If statement block as a question that requires a True or False answer. If the answer is True a section of code your program will execute. Since computers only work with numbers, the answer to the question is an equation that will result in either 0 for False or non-zero for True.

The If statement has the following formats.

```
If <expression> Then Do something[:Do something]
```

The <expression> is the question that requires a True or False answer. If the answer is True the code block following the Then is executed. If the answer is False then the next line of code is executed.

You can execute more than one statement after the Then if you separate them with a colon. The code must be on the same line. An easier format is to use the IF code block, as shown below.

```
If <expression> Then
    Do something 1
    Do something 2
    ...
End If
```

In this format if the answer is True then the code block following the Then is executed. If the answer is False all statements until the End If is reached. The program will then start executing the code following the End If. If the code in the code block is skipped and the code following the End If is executed.

```
If <expression> Then
    Do something
    ...
Else
    Do something Else
```

```
    ...  
End If
```

In this format if <expression> is True then the code following the Then is executed. In this format you can address both the T

```
If <expression> Then  
    Do something  
ElseIf <expression> Then  
    Do something  
End If
```

In this format if <expression> is True then the code following the Then is executed. If the ElseIf is True, the code following the Then (of the ElseIf) is executed. You can have as many ElseIf statements as you need to ful

```
If <expression> Then  
    Do something  
ElseIf <expression> Then  
    Do something  
Else  
    Do something Else  
End If
```

This format is a combination of all the other formats. If <expression> is True the answer is False then the ElseIf is executed. If the ElseIf is True, the otherwise the code following the Else is executed.

This format enables you to ask a series of questions and if the answer is course of action based on the Else block.

As you can see you can frame the question in a number of ways and the

combinations. This gives you a lot of flexibility in how to both frame a qu

The <expression> is the question that needs an answer and you frame t

You can mix arithmetic and logical operators, as well as parenthesis, wit
conditional statements from left to right, taking into account the precede
code snippets are legal If statement constructs.

```
If var1 = 5 Then  
If (var1 = 5) And (var2 < 3) Then  
If (var1 + 6) > 10 Then
```

You will notice that parenthesis are used to group the different parts of th
sure that you are executing logical portions of the expressions. The exp
even if you are using arithmetic operators within the expression.

Using Bitwise Operators in an If Statement

Remember that the operators And, Or and Not are bitwise operators. Th
operation that they perform. You should take care when using bitwise op
result will evaluate correctly.

Take the second code snippet listed above.

```
If (var1 = 5) And (var2 < 3) Then
```

If var1 equals 5, the compiler will return True, or -1 for the expression. If
or -1 for this expression. The compiler will then evaluate the And operat
True, the code following the Then will be executed.

If either of the statements within the parenthesis evaluate to 0, then And
the Then clause will be skipped. When using bitwise operators you shou
the bitwise operator so that they return either True or False. This will giv

The Not Problem

The Not bitwise operator can be a problem in an If statement. You may think of it as a logical, rather than a bitwise operation. In FreeBasic Not performs a bitwise operation.

If var were to contain the value of 3, then Not 3 is -4, which will be regarded as true and the code within the If statement will be executed, which is probably not what you wanted. Instead of writing

Overlapping Conditions

When using compound conditions care must be taken to ensure that the conditions will produce predictable results. Each condition must produce individual results, must itself express a unique result. This is very important within an If-Else block may execute the wrong code at the wrong time.

Nested If Statements

At times it may become necessary to nest If statements in order to better evaluate conditions. While the If statement can handle multiple arguments within a single If statement, it is often easier to incrementally check for certain ranges of values which you can do using

```
If <expression> Then
    <statement>
    ...
    If <expression> Then
        <statement>
        <statement>
        ...
    End If
End If
```

It is important to close each block properly with an End If when opened. Syntax errors are fairly easy to fix, while logical errors can be tricky to track down.

closing the blocks properly is to indent the nested If statements and their level as the If. In the example above, the indentation tells you at a glance

The IIF Function

The IIF, or "immediate If" function returns one of two numeric values based on an in-line If statement that acts as a function call.

```
Value = IIF(<expression>, numeric_value_if_true, numeric_value_if_false)
```

IIF can be used as a standalone function or inside other expressions where an If statement. The numeric values can be literal values, variables or numeric expressions. IIF will only return a numeric value, not a string value, however you can wrap the numeric values in a string function.

The IIF statement will evaluate both the True and False conditions so you need to take care that the conditions are correct, even if that condition is not returned from the function.

Framing the Question

The If statement is a powerful tool, but you need to make sure that you use it correctly. Each expression must resolve to True or False, with True always being the default.

When writing an If statement you must ask yourself, does this expression resolve to True or False? If you have compound expressions that have a number of terms within the expression, the sum of the terms must resolve to True or False. If there is any doubt that the expression will resolve to True or False, use nested If statements.

Checking For Range Values

Often times you will need to check for a range of values within an If statement. To check for a range condition correctly, you must frame the expressions correctly. There are two types of ranges: exclusive and inclusive ranges. Exclusive range expressions exclude a value from the range, while inclusive range expressions include a value from the range.

range of values. Each has a particular format that must be followed for p

Excluding a Range of Values

Suppose that you have a range of values and you want to do something value is greater than or equal to 10. To put this another way, you want to action.

You can frame this as a question that can then be translated into code.

```
Is the value a number less than Or equal To 1 Or a r  
If Yes, Then Do special action.  
If No, Then Do standard action.
```

The key here is the OR. If the lower bound of the value is equal to or less or greater than 10 then do the special action.

```
If (value <= 1) Or (value >= 10) Then  
    do_special  
Else  
    do_standard  
End If
```

Remember that OR will return True if either condition is True. If the value expression will return True and the special action will be performed.

Including a Range of Values

Inclusion is the opposite of exclusion. As you might guess, the format is opposite of the OR operator.

Suppose you want to do something special if the value is a 5, 6 or 7. This range expression. Again, you can start by asking a question.

```
Is the value a number between 5 And 7 (inclusive)?  
If Yes, Then Do special action  
If No, Then Do standard action
```

Here you want to include the numbers 5, 6, 7 for consideration. That is if then do something special. This translates to the following code snippet.

```
If (value >= 5) And (value <= 7) Then  
    do_special  
Else  
    do_standard  
End If
```

Remember that the And operator will only return True if both operands are also less than 7, so both statements are True and the expression evaluates

The Select Case Statement



Written by *rdc*

The Select Case block can be viewed as an optimized If-Elseif ladder, a much the same way. The standard Select Case can use any of the standard operators for <expression> and the specialized Select Case As Const format is optimized for integer values.

This code snippet shows the syntax of the standard select case. Expressions can be a variable which can be of any of the standard data types, or individual elements of a Type or array.

```
Select Case <expression>
  Case <list>
    <statement>
    <statement>
    ...
  Case Else
    <statement>
    <statement>
    ...
End Select
```

The <list> clause of the Case statement can be any of the following forms:

- Case <value>: Value is one of the supported data types or an enumeration.
- Case <value> To <value>: Specifies a range of values.
- Case Is <operator> <value>: Operator is any of the logical operators.
- Case <value>, <value>, ...: List of values separated with commas.
- Case <variable>: A variable that contains a value.

The following snippet illustrates how these different formats may be used:

```
Case 47
```

```
Case 47 To 59
```

```
Case Is > 60
```

```
Case 47, 48, 53
```

```
Case keycode
```

The Select Case As Const is a faster version of the Select statement de with integer expressions in the range of 0 to 4097.

```
Select Case As Const <integer_expression>  
  Case <list>  
    <statement>  
    <statement>  
    ...  
  Case Else  
    <statement>  
    <statement>  
    ...  
End Select
```

The <list> statement formats for the Select Case As Const are limited to enumerations of values. That is, the operator expressions are not allowed Case As Const.

When a Case block is executed, the statements following the Case keyword next Case keyword (or End Select) will be executed. Only one block of statements within a Case will execute at any one time. If a Case Else is present, the statements within the Else block will execute if no Case matches the <expression> portion of the Select statement. The following program illustrates using the Case statement block.

```
'Ascii code of key press
```

```

Dim As Integer keycode

'Loop until esc key is pressed
Do
    keycode = Asc(Inkey)
    Select Case As Const keycode
        Case 48 To 57
            Print "You pressed a number key."
        Case 65 To 90
            Print "You pressed an upper case letter"
        Case 97 To 122
            Print "You pressed a lower case key."
    End Select
    Sleep 1
Loop Until keycode = 27 '27 is the ascii code for Esc

End

```

In the program, when you press a key, the value is translated to a number by the Asc function. Since this will always be an integer value that is less than 256 (the range of ASCII character codes from 0 to 255), the Select Case as Const for

The compiler will check the value of keycode against the Case ranges to determine which block should execute. If keycode falls within a particular range, the Print statement will execute, and then the flow of the program will continue with the next line following the End Select. If keycode doesn't match any Case range, then the program will continue with the next line following the End Select.

A Select Case can usually be translated from an If-Elseif ladder. To illustrate, the previous program is shown below as an If-Elseif ladder.

```

'Ascii code of key press
Dim As Integer keycode

'Loop until esc key is pressed
Do

```

```
keycode = Asc(Inkey)
If (keycode >= 48) And (keycode <= 57) Then
    Print "You pressed a number key."
ElseIf (keycode >= 65) And (keycode <= 90) Then
    Print "You pressed an upper case letter key."
ElseIf (keycode >= 97) And (keycode <= 122) Then
    Print "You pressed a lower case key."
End If
Sleep 1
Loop Until keycode = 27 '27 is the ascii code for Esc
End
```

If you compare the two programs, you can see that the logic is quite similar. The Select Case is much more readable and understandable than the If-Then-ElseIf-End If.

Conditional Compilation And You



Written by aetherFox for *QB Express Issue #9*

Conditional Compilation is one of those parts of programming that sit in the dusty corners of the knowledge banks of programmers world-over, yet is one of the most ingenious additions to any language. Usually something that was reserved for C programmers, with the power of freeBASIC's new preprocessor, you can now use conditional compilation to help your program.

The preprocessor allows you flexibility in changing the way code is generated through the use of conditional compilation. Take this scenario you are debugging the code in your program, and you want to add some extra code to output a few variables, but remove them in the final version. The code would be something like this:

```
#define DEBUG

#ifdef DEBUG
    Print "Debug Value"
#endif 'DEBUG
```

Note you do not need the comment after the `#endif`, but is it good practice.

Basically, the above code checks to see whether `DEBUG` has been defined and if it has, then the code between the `#ifdef...#endif` will be executed. While this may seem silly, the uses this has are amazing. If you simply remove one line at the top of your program (`#define DEBUG`), then all the 'debug code' that you've added won't be sent to the compiler - the preprocessor removes it, reducing the bloat of the final executable.

```
'Turn on debugging
```

```
#define DEBUG

'Turn off debugging
#undef DEBUG
```

The `#undef` directive is a way of 'undefining' something, in this case `DEBUG`. While it is strictly not needed (just commenting out the line `'#define DEBUG`' is enough), it makes the code much clearer, and has other uses:

```
#ifndef DEBUG
    Print "Production Version"
#endif 'DEBUG
```

While not the most useful example, this demonstrates the use of another directive: `#ifndef`. This directive will cause the code to be compiled if the symbol is not defined.

Much like a normal programming language, the sense of the conditional can be reversed using a variant of else, `#else`:

```
#ifdef DEBUG
    Print "Test Version"
#else
    Print "Production Version"
#endif 'DEBUG
```

Of course, there are many applications to this. Who says you need to do this on debug code only? You could actually check the effect of a new piece of code, or some test routines by simply defining a name like

TESTCODE and using the preprocessor directives to encompass your code for conditional compilation:

```
#define TESTCODE

#ifdef TESTCODE
    BulletRoutine()
    TestFireRoutine()
#endif
```

The scope of this tutorial is a limited one, but this method is used by professionals. It makes life easy when programming. I have used this method in my own code. To see this code in action, view the source [her](#)

Avinash 'aetherFox' Vora
avinashvora [at] **gmail** [dot] **com**.
<http://avinash.apeshell.net>

Introduction to Pointers



Written by *rdc*

What is a Pointer?

A pointer is a 4-byte data type that holds an address to a memory location. Once it contains data, it points to data once it has been initialized. An uninitialized pointer contains nothing and is undefined.

To understand pointers, think of an egg carton that has numbers 1 through 12 on the bottom of each "hole" (where you put the eggs). These holes are like memory locations in a computer; each hole, or memory location, has an address, in this example, the number. If an egg represents a data item, then an egg in hole 1 has an address of 1.

Normally, you would access the data directly through the use of a variable. For example, if you have a variable of a particular type, you are setting aside storage for it. You do not need to know, or care, where the data resides since you can access it directly through the variable. This is like reaching out and picking up the egg (reading the data) or putting an egg in hole 1 (setting the data) without looking at the numbers written on the bottom of the hole.

Using pointers is a bit different. Imagine you have a little scrap of paper that you use as our pointer. Right now it is blank and doesn't point to anything. This pointer can only be used until it is initialized. To initialize the pointer, write a 1 on it. Now the pointer is "pointing" to hole 1 in our egg carton. To put data (an egg) in hole 1, we take the egg, match it to hole 1 and place the egg in the hole. To retrieve the egg, we take a slip of paper opposite. We match our slip of paper to hole 1 and then grab the egg. All the getting of the egg has to be done through the slip of paper and is called using a pointer. That is, we get to the data through the reference contained in the pointer, the number 1. The pointer doesn't contain the data; it contains a reference to the data.

In FreeBasic we define a pointer using the **Dim** and **Ptr** statements:

```
Dim aptr As Integer Ptr
```

This statement corresponds to our blank piece of paper in the above example. It doesn't point to anything and is undefined. If we tried to use the pointer, it is likely the program would crash.

In order for a pointer to be useful, it must be initialized:

```
Dim aptr As Integer Ptr  
  
aptr = Allocate(NumberOf(Integer))
```

Here we are using **Allocate** to set aside enough space in memory for an integer and loading the address of that space into `aptr`. The **NumberOf** macro returns the size of the passed data type. You could use `len` instead of **NumberOf** (since `len` is available in .NET).

Once we have initialized the pointer, we can now use it:

```
*aptr = 5  
Print "aptr: "; *aptr
```

Notice the `*` prefix on `aptr`. The `*` is the reference operator. This is like moving a slip of paper to the number on the hole in the egg carton. By using `*aptr`, we are able to get at the data (egg) contained in the hole pointed at by `aptr`.

Here is a complete example program:

```
Option Explicit  
  
Dim aptr As Integer Ptr  
  
aptr = Allocate(NumberOf(Integer))  
*aptr = 5
```

```
Print "aptr: "; *aptr
Deallocate aptr
Sleep
```

The **Deallocate** function frees the memory pointed at by `aptr`, and make once again. This is like erasing the number on our slip of paper. If we we deallocating it, the program would crash.

What Good are Pointers?

A major reason for adding pointers to FreeBasic is that many external libraries use pointers to type structures and pointers to strings. For example, the Windows API has many structures that must be filled out and then passed to a function through a pointer.

Another use of a pointer is in a **Type** definition. **Type** defs in FreeBasic can only be for fixed length strings, but what if you don't know the length of a string until runtime? A pointer can serve this purpose.

(It should be stated that the Type definitions can now support variable length strings.)

Option Explicit

```
Type mytptr
    sptr As ZString Ptr
End Type
```

```
'This function will allocate space for the passed string
'and load it into a memory location, returning the
'pointer to the string.
```

```
Declare Function pSetString(ByVal s As String) As ZString Ptr
```

```
'type var
Dim mytype As mytptr
```

```
'Set a variable string into the type def
mytype.sptr = pSetString("Hello World From FreeBasic")
```

```

Print "aptr: "; *mytype.sptr
Deallocate(mytype.sptr)
Sleep
End

Function pSetString(ByVal s As String) As ZString Ptr
    Dim sz As ZString Ptr

    'allocate some space + 1 for the chr(0)
    sz = Allocate(Len(s) + 1)
    'load the string into the memory location
    *sz = s
    'return the pointer
    Return sz
End Function

```

Here we define our type with a field `sptr` as **ZString Ptr**. Zstrings are null-terminated and are used by many external libraries and are designed for dynamic allocation. To define our type we create an instance of it with the **Dim** statement:

```
Dim mytype As mytptr
```

We then call our function `pSetString` to get the address of the variable `le` in our **Type** def.

```
mytype.sptr = pSetString("Hello World From FreeBasic")
```

Remember `sptr` is defined as a pointer, not a string variable, so `pSetString` returns a pointer (memory address) to the string not the string itself. In other words

hole #1, pSetString returns 1.

The function pSetString uses a temporary **ZString** sz, to **Allocate** space for string parameter s. Because a **ZString** is a null terminated string, we must add the length of s for the null terminator in the **Allocate** function.

```
'allocate some space + 1 for the chr(0)
sz = Allocate(Len(s) + 1)
```

Once we have allocated space for the string, we use the reference operator to load data into the memory location.

```
'load the string into the memory location
*sz = s
```

We then return a pointer (the address of the string) back to our type, which is mytype.sptr.

```
'return the pointer
Return sz
```

We can now reference the string in our type using the reference operator.

```
Print "aptr: "; *mytype.sptr
```

Pointers can be confusing for the uninitiated, however they need not be that the pointer doesn't contain data, it simply points to some data. The address, and you manipulate that data through the reference operator *. different than a normal variable.

Pointers, Data Types and Memory



Written by *rdc*

If you read the article [Introduction to Pointers](#) you know that pointers contain memory location addresses. You can manipulate the data in these memory locations using the reference operator *. Using pointers with single data item isn't a problem, but what if you need to store multiple data items together and manipulate them using a pointer? It can get a bit tricky unless you understand how data is stored in memory.

A single memory location in a computer is 1 byte long. Big enough to hold a single ANSI character (as opposed to Unicode characters, which are wide characters and are two bytes. We won't be discussing Unicode characters in this article.) However, all data types are not a single byte in width. Here is a simple program that displays the length in bytes of each data type.

```
Dim a As Byte
Dim b As Short
Dim c As Integer
Dim d As LongInt
Dim au As UByte
Dim bu As UShort
Dim cu As UInteger
Dim du As ULongInt
Dim e As Single
Dim f As Double
Dim g As Integer Ptr
Dim h As Byte Ptr
Dim s1 As String * 10 'fixed string
Dim s2 As String      'variable length string
Dim s3 As ZString Ptr 'zstring

s1 = "Hello World!"
s2 = "Hello World from FreeBasic!"
s3 = Allocate( Len( s2 ) + 1 )
```

```
*s3 = s2

Print "Byte: ";Len(a)
Print "Short: ";Len(b)
Print "Integer: ";Len(c)
Print "Longint: ";Len(d)
Print "UByte: ";Len(au)
Print "UShort: ";Len(bu)
Print "UInteger: ";Len(cu)
Print "ULongint: ";Len(du)
Print "Single: ";Len(e)
Print "Double: ";Len(f)
Print "Integer Pointer: ";Len(g)
Print "Byte Pointer: ";Len(h)
Print "Fixed String: ";Len(s1)
Print "Variable String: ";Len(s2)
Print "ZString: ";Len(*s3)

Deallocate s3

Sleep
```

The output is:

```
Byte: 1
Short: 2
Integer: 4
LongInt: 8
UByte: 1
UShort: 2
UInteger: 4
ULongInt: 8
Single: 4
Double: 8
Integer Pointer: 4
```

```
Byte Pointer: 4
Fixed String: 10
Variable String: 27
ZString: 27
```

Notice that the length of a pointer is always 4 bytes long (the same as an integer), regardless of the data being pointed to, since a pointer contains a memory address and not data.

Looking at the length of the different data types, you can see that if you were to **Allocate** enough space for 10 integers, it would take 40 bytes of memory. Each integer takes up 4 bytes. So the question is, how do you access each integer value from the memory buffer? The answer, pointer math. Take a look at the following program.

Option Explicit

```
Dim a As Integer
Dim aptr As Integer Ptr

'Allocate enough space for 2 integers
aptr = Allocate(Len(a) * 2)
'Load our first integer
*aptr = 1
Print "Int #1: ";*aptr
'Move the pointer to the next integer position
'aptr + 4
*(aptr + 4) = 2
Print "Int #2: ";*(aptr + 4)

Deallocate aptr
Sleep
End
```

In this program we dimension two variables, an **Integer** and an **Integer Pointer**, `aptr`. `aptr` will point to our memory buffer that will contain two integers. The **Allocate** function requires the size of the buffer we need, so we multiply the size of an **Integer** by 2 to reserve 8 bytes of memory (each integer will take 4 bytes of space).

After the allocation process, `aptr` contains the address of the first byte of our memory buffer. Storing the first integer is simply a matter of using the reference operator and setting the value to 1. To print out the value, we again just use `*aptr`.

Now, let me ask you a question: How does the compiler know that the value 1 requires 4 bytes and not 1 or 2 bytes? Because we dimensioned `aptr` as an *integer ptr*. The compiler knows that an integer takes 4 bytes and so loads the data into four bytes of memory. This is why when we print out the value we get 1 and not some strange number.

To load the second value into our buffer, we use:

```
*(aptr + 4) = 2
```

This may look a little strange at first glance. `aptr` points to the first byte in our memory buffer. An integer is 4 bytes long, so to get to the next integer byte position, we must add 4 to `aptr`. We need the parenthesis around the add operation because the reference operator `*` has a higher precedence than `+`. The parenthesis ensure that we perform the add operation first, and then apply the indirection operator.

Notice that we didn't increment `aptr` directly. If we did, `aptr` would no longer point to the start of the memory buffer and the program would crash when we deallocated the buffer since it would **Deallocate** memory outside the memory buffer. If the need arises to directly increment a pointer, then create a temporary pointer variable and increment that,

rather than the pointer used in the original allocation.

Memory buffers and pointers are a powerful way to store and manipulate data in memory. Care must be taken though to ensure that you are accessing the data correctly according to the type of data being stored in the buffer.

The Pointer Data Type



Written by *rdc*

The pointer data type is unique among the FreeBasic numeric data type

On a 32-bit system, the pointer data type is 4 bytes. FreeBasic uses pointers fast, since the compiler can directly access the memory location that a pointer

For many beginning programmers, pointers seem like a strange and mysterious concept. A pointer contains an address, not data. If you keep this simple rule in mind, you can

Pointers and Memory

You can think of the memory in your computer as a set of post office boxes. You decide to write the number down on a slip of paper and put it in your mailbox. In other words, of course, you want to toss the junk mail, but there isn't a trash can handy,

When you declare a pointer, it isn't pointing to anything which is analogous to a slip of paper. Once you have the address, find the right P.O. Box, you can access the data at pointers.

Declare a pointer variable.

Initialize the pointer to a memory address.

Dereference the pointer to manipulate the data at the pointed-to memory location.

This isn't really any different than using a standard variable, and you use pointers directly, and with a pointer you must dereference the pointer to interact with the data.

Typed and Untyped Pointers

FreeBasic has two types of pointers, typed and untyped. A typed pointer

```
Dim myPointer As Integer Ptr
```

This tells the compiler that this pointer will be used for integer data. Using arithmetic.

Untyped pointers are declared using the Any keyword.

```
Dim myPointer As Any Ptr
```

Untyped pointers have no type checking and default to size of byte. Untyped pointers specifically need an untyped pointer, you should use typed pointers so that the compiler can check the data.

Pointer Operators

FreeBasic has the following **pointer operators**.

You will notice that the addressof operator not only returns the memory address of a variable, but also a function such as used in the CRT QSort function.

Memory Functions

FreeBasic also has a number of **memory functions** that are used with pointers.

These functions are useful for creating a number of dynamic structures such as arrays.

When using the Allocate function you must specify the storage size based on the number of elements. Allocate(10 * Sizeof(Integer)). An integer is 4 bytes so allocating 10 integers will be 40 bytes and initialized.

Callocate works in the same fashion, except that the calculation is done for you. Callocate will clear the memory segment.

Reallocate will change the size of an existing memory segment, making a new segment if the existing segment is smaller than the existing segment, the data in the existing segment will be copied to the new segment.

All of these functions will return a memory address if successful. If the function returns 0, be sure that the memory segment was successfully created. Trying to use a pointer that is 0 will cause a runtime error.

There is no intrinsic method for determining the size of an allocation. You must keep track of this.

Be careful not to use the same pointer variable to allocate two or more memory segments. Reuse

Pointer Arithmetic and Pointer Indexing

When you create a memory segment using the allocation functions, you use the dereference operator with pointer arithmetic, and pointer indexing.

Pointer arithmetic, as the name suggests, adds and subtracts values to a pointer that the data being used with this pointer is of size Integer or 4 bytes. This is done by adding 1 to the pointer, which can be expressed as `*(myPtr + 1)`.

Since the compiler knows that the pointer is an Integer pointer, adding 1 works for untyped pointers. The compiler does much of the work for you in accessing

Notice that the construct is `*(myPtr + 1)` and not `*myPtr + 1`. The `*` operator

`myPtr` will be evaluated first, which returns the contents of the memory location. Then `+ 1` first, which increments the pointer address, and then the `*` is applied to

Pointer indexing works the same way as pointer arithmetic, but the details are more correct memory offsets to return the proper values using the index. Which is done by the dereference operator.

Pointer Functions

Freebasic has a set of pointer functions to complement the pointer operators.

CPtr Converts expression to a `data_type` pointer. Expression can be any integer

Peek Returns the contents of memory location pointed to by pointer. Data type is

Poke Puts the value of expression into the memory location pointed to by pointer.

SAdd Returns the location in memory where the string data in a dynamic array is stored.
StrPtr The same as SAdd.
ProcPtr Returns the address of a function. This works the same way as the addressof operator @.
VarPtr This function works the same way as the addressof operator @.

The SAdd and StrPtr functions work with the string data types to return the address of the data, like the address of operator @, but ProcPtr only works on subroutines and functions.

Subroutine and Function Pointers

Subroutines and functions, like variables, reside in memory and have an address. You create a sub or function pointer just like any other pointer.

Before using a function pointer, it must be initialized to the address of a subroutine or function.

You declare a function pointer using the anonymous declaration syntax.

```
Dim FuncPtr As Function(x As Integer, y As Integer)
```

You then need to associate this function pointer with an actual subroutine or function.

```
Function Power(number As Integer, pwr As Integer) As Integer  
Return number^pwr  
End Function  
  
FuncPtr = @Power
```

You can then call the function pointer much like you would call the real function.

```
FuncPtr(2, 4)
```

While this may not be useful at first glance, you can use this technique to

For example, suppose you have a dog and cat object. Both objects need to make Speak either issue a "Woof!" or "Meow!" depending on the object.

Creating a Callback Function

One of the primary uses for function pointers is to create callback functions from an external library. Windows uses callback functions to enumerate through

The qsort, function contained within the C Runtime Library sorts the elements

```
Declare Sub qsort cdecl Alias "qsort" (ByVal As Any
```

The following lists the parameter information for the qsort subroutine.

The first parameter is the address to the first element of the array. The second parameter is the number of elements in the array, that is the total size of the array. The third parameter is the size of each element in bytes. For an array of integers, the size is 4 bytes. The fourth parameter is a function pointer to the user created compare function.

Using this information, you can see how qsort works. By passing the address of the array and the number of elements, qsort will take two array elements, pass them to your user defined compare function, and return the result.

Qsort will take two array elements, pass them to your user defined compare function, and return the result.

You need to declare the function prototype as cdecl which ensures that the function is called in C style.

```
Declare Function QCompare cdecl (ByVal e1 As Any Ptr
```

You would then define the function like the following.

```

'The qsort function expects three numbers
'from the compare function:
'-1: if e1 is less than e2
'0: if e1 is equal to e2
'1: if e1 is greater than e2
Function QCompare cdecl (ByVal e1 As Any Ptr, _
ByVal e2 As Any Ptr) As Integer
Dim As Integer e11, e12
Static cnt As Integer

'Get the call count and items passed
cnt += 1
'Get the values, must cast to integer ptr
e11 = *(CPtr(Integer Ptr, e1))
e12 = *(CPtr(Integer Ptr, e2))
Print "Qsort called";cnt;" time(s) with";e11;" and",
'Compare the values
If e11 < e12 Then
Return -1
ElseIf e11 > e12 Then
Return 1
Else
Return 0
End If
End Function

```

You would then call the QSort function passing the address of the callba

```
qsort @myArray(0), 10, SizeOf(Integer), @QCompare
```

Pointer to Pointer

In FreeBasic you can create a pointer to any of the supported data types structures such as linked-lists and ragged arrays. A pointer to a pointer i

One application of a pointer to pointer is the creation of a memory segment dynamic memory segment that you can resize as needed during runtime

```
Dim myMemArray As Integer Ptr Ptr
```

You would then initialize the pointer reference by using `Allocate` or `Call`

```
'Create 10 rows of integer pointers  
myMemArray = CAllocate(10, SizeOf(Integer Ptr))
```

Notice that the variable is initialized to an *Integer Ptr* since this list is going to hold needed memory segments.

```
'Add 10 columns of integers to each row  
For i = 0 To 9  
myMemArray[i] = CAllocate(10, SizeOf(Integer))  
Next
```

In this code snippet, the individual pointers in the list are initialized to 10

```
'Add some data to the memory segment  
For i = 0 To 9  
For j = 0 To 9  
myMemArray[i][j] = Int(Rnd * 10)  
Next  
Next
```

This code snippet uses the index method to load the actual data into the

code to create a dynamic array within a type definition. Since you cannot

One thing you need to be aware of is how to deallocate a structure such as these memory segments first and then you can deallocate the base pointer

```
'Free memory segment
For i = 0 To 9
Deallocate myMemArray[i]
Next

'Free the pointer to pointer
Deallocate myMemArray
```

You need to be sure that you deallocate in the right order, otherwise you

Linked Lists



A linked list is a structure that is easily expandable by using a single function of something but you have no idea how many. The concept behind a linked list is a node and previous node structure. This is called a double linked list, as it links you can specify a null pointer if there is no next or previous node, and since the number of nodes you can store is limited only by memory.

The only downside to using a linked list is that in order to store say an integer but also a structure that contains a pointer to the integer and a pointer to the next node. On today's computers however, unless you are storing millions of nodes, this is not a problem.

The basic structure of the linked list is the node. The declaration is this:

```
Type listnode
  As Any Ptr pData
  As listnode Ptr pNext
  As listnode Ptr pPrev
End Type
```

As a side note, if whoever has access to these scripts would like to update the code (as ptr), feel free to :) Also, LIST doesn't appear to be an FB keyword (could be).

This structure contains three pointers. The first is a pointer to anything (could be a character, even user defined types and unions. But it also means that you can use the Allocate (or CAllocate) function.

The next two pointers are pointers to listnodes, that is, you are technically printing node->pNext->pNext->pNext->pNext->pNext... since each node contains a pointer to another node. The problem with this is that you can access and the code gets hard to understand. You can use the While loop.

Before we go any further, let's see all the declarations for using linked lists.

```
Declare Function ListCreate() As listnode Ptr
Declare Function ListAdd(list As listnode Ptr, item As Any) As listnode Ptr
Declare Function ListAddHead(list As listnode Ptr, item As Any) As listnode Ptr
```

```

Declare Function ListGetFirst(list As listnode Ptr)
Declare Function ListGetLast(list As listnode Ptr) /
Declare Function ListGetNext(list As listnode Ptr) /
Declare Function ListGetPrev(list As listnode Ptr) /
Declare Function ListGetData(list As listnode Ptr) /
Declare Function ListRemove(list As listnode Ptr, bDe:
Declare Sub ListRemoveAll(list As listnode Ptr, bDe:

```

Edit: Hmm, it doesn't seem to like my use of "Rem" in a function. It comp

You can see that there is a function to create a linked list, to add an item
 Currently we'll focus on the ListCreate function. It takes no parameters and
 has no data filled out. The whole structure is null, but it is still a structure
 point to the new item, so it won't stay as a null node, since there would be
 ListCreate won't have any data stored in it and it won't have a previous r

The function ListCreate looks like this:

```

' CREATE
Function ListCreate() As listnode Ptr
    Dim As listnode Ptr pTemp
    pTemp = CAllocate(Len(listnode))
    ' CAllocate automatically zeroes memory.

    Return pTemp
End Function

```

I prefer to use the Return instruction to return a value from a function, but
 allowed, although they don't immediately exit the function.

The point of this function is easy to see, a node is allocated and returned
 automatically zeroes memory. If you used the Allocate function, the mem
 have to do that on your own.

The next functions, ListAdd and ListAddHead, add a node to the list. Lis
 ListAddHead puts a node at the very top (the head).

```

' ADD, ADDHEAD

Function ListAdd(list As listnode Ptr, item As Any Ptr) As listnode Ptr
    Dim As listnode Ptr pTemp

    If (list = 0) Then Return item

    pTemp = ListGetLast(list)

    pTemp->pNext = CAllocate(Len(listnode))
    pTemp->pNext->pPrev = pTemp
    pTemp->pNext->pData = item

    Return item
End Function

Function ListAddHead(list As listnode Ptr, item As Any Ptr) As listnode Ptr
    Dim As listnode Ptr pTemp

    If (list = 0) Then Return item

    pTemp = list->pNext
    list->pNext = CAllocate(Len(listnode))

    list->pNext->pPrev = list
    list->pNext->pData = item
    list->pNext->pNext = pTemp

    If (pTemp <> 0) Then
        pTemp->pPrev = list->pNext
    End If

    Return item
End Function

```

You can see that ListAdd makes a reference to a function not shown yet and returns a pointer to the last node in the list. It will be covered later.

ListAdd retrieves the last node and sets its pNext pointer to a new listnode. The last node has a null pNext value because nothing comes after it. Once our new node is added, the line

```
pTemp->pNext->pPrev = pTemp
```

is the whole basis of linked lists, the linking part. What this says is that the next node is, and now we're telling the node after that next one where the next node is. The compiler doesn't know where the nodes are until you set them. Once

The ListAddHead function is a little more complicated, since we're actually adding a new node to the beginning of the list. It starts by creating a new null node from ListCreate. What it does basically is allocates space to hold the new node and links them all together. If you study it a little, it should seem a lot clearer. It's just trying to access memory that doesn't exist (NULL->pPrev). If pTemp does not exist, it's not assigned. Otherwise, there is no reason to worry about it.

The next functions are ListGetFirst and ListGetLast. I implemented them as follows.

```
' GETFIRST, GETLAST

Function ListGetFirst(list As listnode Ptr) As listnode Ptr
    If (list = 0) Then Return 0

    Return list->pNext
End Function

Function ListGetLast(list As listnode Ptr) As listnode Ptr
    Dim As listnode Ptr pTemp

    If (list = 0) Then Return 0

    pTemp = list
    While (pTemp->pNext <> 0)
        pTemp = pTemp->pNext
    Wend

    Return pTemp
End Function
```

The first function is probably the shortest and easiest function to understand. It returns a pointer to the node returned by ListCreate. If you don't do this, it could return the first node, or the node that comes right after the null node.

The second function, ListGetLast, loops through the list until it finds a null node. The only change I made to the original implementation of pTemp = 0 is that I don't want to return zero. I want to return the last non-null node. Once that node is found, ListGetLast returns it.

The next 3 functions are just helper functions, and could be easily added to the original implementation not written by me had a ListGetNext function.

```
' GETNEXT, GETPREV

Function ListGetNext(list As listnode Ptr) As listnode Ptr
    If (list = 0) Then Return 0

    Return list->pNext
End Function

Function ListGetPrev(list As listnode Ptr) As listnode Ptr
    ' can't do anything to a null list
    If (list = 0) Then Return 0
    ' this is needed for below
    If (list->pPrev = 0) Then Return 0
    ' since the list starts with a null node (pPrev = 0)
    ' the first should be the one right after the null node
    If (list->pPrev->pPrev = 0) Then Return 0

    Return list->pPrev
End Function

' GETDATA

Function ListGetData(list As listnode Ptr) As Any Ptr
    If (list = 0) Then Return 0
```

```
Return list->pData
End Function
```

The first function, ListGetNext, is the exact same as ListGetFirst, but the ListGetFirst on a node value in this implementation, it isn't a smart idea to begin of the list in order to find the first node, in which case you'd be

The ListGetPrev function is a little more complicated, since I don't want the comments) are the ones that are actually needed, but the second one explains that if two nodes up is null, we should return zero. That means that previous node that you can do anything with, although there does exist a handles the default case, where there is in fact a previous node, and it s

The ListGetData function is as easy and brief as the ListGetFirst and List data.

The final two functions remove nodes from the list.

```
' REMOVE, REMOVEALL

Function ListRemove(list As listnode Ptr, bDelete As
    Dim As listnode Ptr pPrev
    Dim As listnode Ptr pNext

    If (list = 0) Then Return 0

    pPrev = list->pPrev
    pNext = list->pNext

    If ((list->pData <> 0) And (bDelete <> 0)) Then

        Deallocate list

    If (pPrev <> 0) Then
        pPrev->pNext = pNext
    End If
    If (pNext <> 0) Then
```

```

        pNext->pPrev = pPrev
    End If

    Return pNext
End Function

Sub ListRemoveAll(list As listnode Ptr, bDelete As Boolean)
    Dim As listnode Ptr node

    node = list
    If (list = 0) Then Return

    While (node <> 0)
        If ((node->pData <> 0) And (bDelete <> 0)) Then
            ListRemove(node)
        End If
        node = node->pNext
    Wend
End Sub

```

The ListRemove function has two jobs: To remove the node you specify and to update the pointers. You can see that it stores a previous and next pointer to do this. The optional parameter bDelete indicates whether the data should be deleted. If you are just storing integers, or even structures with no pointers, the ListRemove will delete the data for you. But if you have a structure with pointers in it, the ListRemove only handles the list part to ensure that there is no memory leak, whether or not you told it to delete the data.

ListRemoveAll relies on the ListRemove function to delete the nodes. It originally deleted every node. The original code used a For loop, but FB doesn't support For node = list To 0 Step ListRemove(node) so it has been changed.

That's it, here's the whole file that includes a sample at the top of how to use it. Feel free to leave comments on ways I could improve. Also, if you catch a bug, let me know. Feel free to edit the bug out also, but I'd like to know about it.

```

Type listnode
    As Any Ptr pData

```

```

    As listnode Ptr pNext
    As listnode Ptr pPrev
End Type

Declare Function ListCreate() As listnode Ptr
Declare Function ListAdd(list As listnode Ptr, item
Declare Function ListAddHead(list As listnode Ptr, :
Declare Function ListGetFirst(list As listnode Ptr)
Declare Function ListGetLast(list As listnode Ptr) /
Declare Function ListGetNext(list As listnode Ptr) /
Declare Function ListGetPrev(list As listnode Ptr) /
Declare Function ListGetData(list As listnode Ptr) /
Declare Function ListRemove(list As listnode Ptr, bDe
Declare Sub ListRemoveAll(list As listnode Ptr, bDe

Dim As listnode Ptr list, node
Dim As Integer Ptr item
list = ListCreate()
item = ListAdd(list, CAllocate(Len(Integer)))
*item = 4
item = ListAdd(list, CAllocate(Len(Integer)))
*item = 44
item = 0 ' just to show it works
node = ListGetFirst(list)

While node <> 0
    Print "found item"
    item = ListGetData(node)
    Print *item
    node = ListRemove(node,1)
Wend

While Inkey$ = "" : Wend

' CREATE
Function ListCreate() As listnode Ptr
    Dim As listnode Ptr pTemp
    pTemp = CAllocate(Len(listnode))
    ' CAllocate automatically zeroes memory.

```

```
    Return pTemp  
End Function
```

```
' ADD, ADDHEAD
```

```
Function ListAdd(list As listnode Ptr, item As Any Ptr  
    Dim As listnode Ptr pTemp
```

```
    If (list = 0) Then Return item
```

```
    pTemp = ListGetLast(list)
```

```
    pTemp->pNext = CAllocate(Len(listnode))
```

```
    pTemp->pNext->pPrev = pTemp
```

```
    pTemp->pNext->pData = item
```

```
    Return item
```

```
End Function
```

```
Function ListAddHead(list As listnode Ptr, item As Any Ptr  
    Dim As listnode Ptr pTemp
```

```
    If (list = 0) Then Return item
```

```
    pTemp = list->pNext
```

```
    list->pNext = CAllocate(Len(listnode))
```

```
    list->pNext->pPrev = list
```

```
    list->pNext->pData = item
```

```
    list->pNext->pNext = pTemp
```

```
    If (pTemp <> 0) Then
```

```
        pTemp->pPrev = list->pNext
```

```
    End If
```

```
    Return item
```

```
End Function
```

```
' GETFIRST, GETLAST
```

```
Function ListGetFirst(list As listnode Ptr) As listnode  
    If (list = 0) Then Return 0  
  
    Return list->pNext  
End Function
```

```
Function ListGetLast(list As listnode Ptr) As listnode  
    Dim As listnode Ptr pTemp  
  
    If (list = 0) Then Return 0  
  
    pTemp = list  
    While (pTemp->pNext <> 0)  
        pTemp = pTemp->pNext  
    Wend  
  
    Return pTemp  
End Function
```

```
' GETNEXT, GETPREV
```

```
Function ListGetNext(list As listnode Ptr) As listnode  
    If (list = 0) Then Return 0  
  
    Return list->pNext  
End Function
```

```
Function ListGetPrev(list As listnode Ptr) As listnode  
    ' can't do anything to a null list  
    If (list = 0) Then Return 0  
    ' this is needed for below  
    If (list->pPrev = 0) Then Return 0  
    ' since the list starts with a null node (pPrev = 0)  
    ' the first should be the one right after the root  
    If (list->pPrev->pPrev = 0) Then Return 0  
  
    Return list->pPrev
```

```
End Function
```

```
' GETDATA
```

```
Function ListGetData(list As listnode Ptr) As Any Ptr
```

```
    If (list = 0) Then Return 0
```

```
    Return list->pData
```

```
End Function
```

```
' REMOVE, REMOVEALL
```

```
Function ListRemove(list As listnode Ptr, bDelete As Boolean)
```

```
    Dim As listnode Ptr pPrev
```

```
    Dim As listnode Ptr pNext
```

```
    If (list = 0) Then Return 0
```

```
    pPrev = list->pPrev
```

```
    pNext = list->pNext
```

```
    If ((list->pData <> 0) And (bDelete <> 0)) Then
```

```
        Deallocate list
```

```
        If (pPrev <> 0) Then
```

```
            pPrev->pNext = pNext
```

```
        End If
```

```
        If (pNext <> 0) Then
```

```
            pNext->pPrev = pPrev
```

```
        End If
```

```
        Return pNext
```

```
End Function
```

```
Sub ListRemoveAll(list As listnode Ptr, bDelete As Boolean)
```

```
    Dim As listnode Ptr node
```

```
    node = list
```

```
If (list = 0) Then Return

While (node <> 0)
    If ((node->pData <> 0) And (bDelete <> 0)) Then
        node = ListRemove(node)
    Wend
End Sub
```

If you haven't noticed already, ListAdd and ListAddHead return a pointer shows how to use this functionality. ListRemove returns a pointer to next ListRemoveAll is the only function that doesn't return anything. There is called it.

Dynamic Arrays in Types



Written by *rdc*

Introduction

A dynamic array in a type definition is a very useful feature, but FreeBasic doesn't support it before version 1.00.0. Or rather, it doesn't support it directly before that version. However, you can create dynamic arrays by using pointers and the associated memory functions.

An array is simply a contiguous block of memory that holds a certain data type. Arrays in FreeBasic use an array descriptor to describe the data contained within the array, and you can use this same technique to build a dynamic array within a type. The two elements you need within your type-def are a pointer to a particular data type, and a size indicator.

You can then use the ptr field to allocate a block of memory to the needed size, and save that size in the size indicator field. The size field is used to tell you how many elements are currently in the array. Once the array has been initialized, you can then use pointer indexing to access each element in the array.

Getting the Point(er) in Code

The following program illustrates the steps in creating, initializing and resizing a dynamic type-def array.

```
'Define type:
'size is current size of array
'darray will contain array data
Type DType
    size As Integer
    darray As Integer Ptr
End Type

'Create an instance of type
Dim myType As DType
```

```

Dim As Integer i, tmp

'Create enough space for elements
myType.darray = CAllocate(5, SizeOf(Integer))
'Set the length of the array
'in the array size indicator
myType.size = 5

'Load data into array
For i = 0 To myType.Size - 1
    myType.darray[i] = i
Next

'Print data
For i = 0 To myType.Size - 1
    Print "darray[";i;" ]:";myType.darray[i]
Next
Print "Press any key..."
Sleep
Print

'Save the current array size
tmp = myType.size
'Now resize the array
myType.darray = Reallocate(myType.darray, 10)
'Set the length indicator
myType.size = 10

'Load in data into new allocation
For i = tmp To myType.Size - 1
    myType.darray[i] = i
Next

'Print out contents
For i = 0 To myType.Size - 1
    Print "darray[";i;" ]:";myType.darray[i]
Next
Print "Press any key..."
Sleep

```

```
'Free allocated space  
Deallocate myType.darray  
  
End
```

How it Works

The first step is, of course, to define the type-def:

```
Type DType  
    size As Integer  
    darray As Integer Ptr  
End Type
```

Since this is just an example there are only two elements within the type a size indicator and the array pointer. Notice that the array pointer is defined as an Integer ptr. When you define a pointer to a particular type, you are creating a "typed" pointer. The compiler can use this type information to check to make sure the values being placed into the array are valid, and will also use this information for pointer arithmetic.

The next step is to define the working variables.

```
Dim myType As DType  
Dim As Integer i, tmp
```

Here an instance of the type is created, as well as some working variables that are used in the following code. WARNING: You must initialize the array pointer before you can use it; using an uninitialized pointer can cause program crashes, system lockups and all sorts of bad things.

```
myType.darray = CAllocate(5, SizeOf(Integer))  
myType.size = 5
```

These two lines of code initialize the array pointer to hold 5 integers. `Callocate` is used to allocate the memory segment, since `Callocate` will initialize the segment to zeros.

The size field stores the current length of the array. Now, of course, you could calculate the size of the array by simply dividing the number of bytes in the allocation by the size of an integer, but using a size indicator within the type is much cleaner and saves you a calculation in your program.

```
For i = 0 To myType.Size - 1
    myType.darray[i] = i
Next
```

This section of code loads the array with some values. You can see why saving the size of the array simplifies the coding process. Since the array is a typed pointer, you can access the array using the pointer indexing method, which is almost like accessing a predefined array.

```
For i = 0 To myType.Size - 1
    Print "darray[";i;" ]:";myType.darray[i]
Next
```

This section simply prints out the values using the same method that was used to load the array.

Of course, this should be a dynamic array, so you should be able to resize the array, and this is exactly what the next section of code will do.

```
tmp = myType.size
myType.darray = Reallocate(myType.darray, 10)
myType.size = 10
```

The first line of code saves the current size of the array so that the new memory segment can be initialized while not overwriting any existing

data. You will see this in a moment.

The second line uses the Reallocate function to resize the memory segment, that is, resize the array. In this case, the array is being made larger; you could of course make the array smaller. If you were to make the array smaller, any data not in the new segment would be lost, as you would expect.

The last line of code above saves the new array size in the size indicato

```
For i = tmp To myType.Size - 1
    myType.darray[i] = i
Next
```

Here, you can see why the old array size was saved. In the For statement, the initialization procedure iterates through the newly added indexes, storing data within the memory segment. This is like using the Redim Preserve statement on a normal array.

```
For i = 0 To myType.Size - 1
    Print "darray[";i;" ]:";myType.darray[i]
Next
```

This code section simply prints out the new values.

```
Deallocate myType.darray
```

This is vitally important. You should always deallocate any allocated memory that you have created in your program to prevent memory leaks

When you run the program you should see the following output:

```
darray[ 0 ]: 0
darray[ 1 ]: 1
```

```
darray[ 2 ]: 2
darray[ 3 ]: 3
darray[ 4 ]: 4
Press Any key...
```

```
darray[ 0 ]: 0
darray[ 1 ]: 1
darray[ 2 ]: 2
darray[ 3 ]: 3
darray[ 4 ]: 4
darray[ 5 ]: 5
darray[ 6 ]: 6
darray[ 7 ]: 7
darray[ 8 ]: 8
darray[ 9 ]: 9
Press Any key...
```

The first print out shows the original array. The second print out shows the newly resized array.

From fbc version 1.00.0, dynamic arrays fields as non-static members are supported inside UDT

Previous example transposed for fbc version 1.00.0 or greater, by using dynamic array field as non-static member inside the UDT (feature now supported):

```
'Define type (for fbc version >= 1.00.0):
'darray will contain array data
Type DType
    darray(Any) As Integer
End Type

'Create an instance of type
Dim myType As DType
Dim As Integer i, tmp
```

```
'Create enough space for elements
ReDim myType.darray(4)

'Load data into array
For i = 0 To UBound(myType.darray)
    myType.darray(i) = i
Next

'Print data
For i = 0 To UBound(myType.darray)
    Print "darray(";i;" ):"; myType.darray(i)
Next
Print "Press any key..."
Sleep
Print

'Save the current array upper bound
tmp = UBound(myType.darray)
'Now resize the array
ReDim Preserve myType.darray(10)

'Load in data into new allocation
For i = tmp + 1 To UBound(myType.darray)
    myType.darray(i) = i
Next

'Print out contents
For i = 0 To UBound(myType.darray)
    Print "darray(";i;" ):";myType.darray(i)
Next
Print "Press any key..."

Sleep
```

Function Overloading



written by *:stylin:*

What is It?

Function overloading is as close as you can come to generic programming, the emphasis is on value, while in generic programming, it's the type of the argument passed. Function overloading is a side-step into a world associated with a variety of functions that work with a variety of different

Simply put, function overloading involves defining functions that have the same name but a combination of all the information needed to correctly reference the function's parameter type. These are what we redefine, or overload. Let's start off with a small example of a number. We simply write:

```
Option Explicit          ' force explicit declaration of all variables
Option ByVal             ' default passing convention is ByVal

' to declare functions with similar functionality but different parameter types
' we 'simply' create new function names :(
Declare Function print_byte( As Byte )      ' outputs a byte
Declare Function print_short( As Short )    ' outputs a short

Dim As Byte b = 102
Dim As Short s = 10240

print_byte( b )
print_short( s )

Sleep : End 0

' function definitions squished for brevity - don't
' get too big in a
constrained tutorial ;}
Function print_byte( n As Byte ) : Print Str( n ) :
```

```
Function print_short( n As Short ) : Print Str( n )
```

What Does It Do For Me?

The problem here is that not only do we have two different function signatures - not the compiler - have to remember both in order to call the right function, it's also confusing if you decide you want to support INTEGERS, SINGLES and LONGS. We can have functions that accept both the signed and unsigned versions of each type, but we have a scheme setup to make this easier on yourself. And, of course you'll want to be able to work about pointers. OK, now you'll need to double the list of function names and signatures when you're actually writing code that uses these functions. Since, after all, you're the one you, and the compiler will happily let you slip a DOUBLE in to your print function, there must be a better way?

There is, and don't call me Shirley. I mentioned before that the compiler uses the function signature: the parameter list and the return type. I also mentioned that there are different signatures, and still keep the same function name for all of them. It's a convoluted name space and all. Well, you're right - check this out:

```
Option Explicit          ' force explicit declaration of all variables and
Option ByVal            ' default passing convention is ByVal

' to overload function print_numeric that we can reuse for different
' types while keeping the name intact, we use the Overload keyword
Declare Function print_numeric Overload( As Byte )
Declare Function print_numeric( As Short )
Declare Function print_numeric( As Integer )
Declare Function print_numeric( As LongInt )

' define some variables
Dim As Byte b = 102
Dim As Short s = 10240
```

```

Dim As Integer i = 1024000000
Dim As LongInt li = 10240000000000000000

' enter the wonderful world of function overloading
print_numeric( b )
print_numeric( s )
print_numeric( i )
print_numeric( li )

Sleep : End 0

' define our function overloads
Function print_numeric( n As Byte ) : Print Str( n )
Function print_numeric( n As Short ) : Print Str( n )
Function print_numeric( n As Integer ) : Print Str( n )
Function print_numeric( n As LongInt ) : Print Str( n )

```

What does It Mean?

One thing that should stand out right away is how incredibly easy it is to add new features to your code while maintaining flexibility and *type-safety* if offers you, but then again most higher-level languages will not only make your life a whole lot easier, but you'll be spending less time writing code of code you write.

It means *flexibility*. Function overloading offers the ability to add more features to your code without breaking current code intact. Your code doesn't break because you want to support a new feature, or whatever else. You may now be thinking that the above code is a bit messy, but it is - is really the foundation of writing better code. You'd be right.

It means *maintainability*: So you've got your 80 functions of `print_some_long_name_you_need_to_look_up_everytime_you_want_to` in your code, it's a torturous, self-loathing world. What happens when something needs to be changed? BAM! A maintenance nightmare. You're going to have to search the entire codebase for every instance of that function name.

function here or there; sad way to spend a Saturday night, my friend.

It means *safety*: You may notice that I utilize two `OPTIONS` in these examples and I'm even bigger on having the compiler watch my back for me. I use `get`. Function overloading also affords you safety - safety against evil (re-actually returning a value from these functions that was dependent on the allowed to get truncated without our knowledge, that spells many pills of pain). It's all about the type-safety, something which causes many to scoff at C++.

Wrapping Up

I hope you have learned at least the basics of function overloading (since the themes I've brought up, if you haven't before). Next time I'll discuss overloading with different return types, as well as the joys and pitfalls of both. Stay tuned.

(a.k.a. stylin)

FreeBASIC

I regularly use and recommend FreeBASIC to anyone needing a language that provides ease-of-use, low development times, portability and support for a variety of programming paradigms. I log on occasionally at the official FreeBASIC site [www.freebasic.net] and read about what's new with FreeBASIC and its great community at the forum: there [www.freebasic.net/forum].

contact

Reach me via email at gmail.com with a username of "laananfisher".

Introduction to Message-Based Programming



Written by rdc

Historically, programming languages have been categorized as procedural. QuickBasic could be categorized as a procedural language and Visual Basic as event-driven (or event-driven) language. In a procedural language you generate code in a somewhat linear manner. In a message-based language, your program sits in an idle loop and waits for something to happen. When something happens, the program returns to the idle loop, eventually exiting the loop when the

In a procedural language you have full control over what the user sees and how it works in cooperation with the operating system and user, handling only the user input and letting the operating system handle the rest. The real stumbling block in moving from a message-based language to a procedural language is the concept of control flow. We are really talking about shades of gray, rather than black and white. In message-based languages, messages play an important role.

If you have ever used a language that supports subroutine and function programming. For example, say you have written a game in QuickBasic that uses arrow keys to be pressed. If the up arrow key is pressed, you call a subroutine to move a sprite on the screen. If the A key is pressed, you ignore it, since you don't care about the A key in your programming. The message is the key press and the sprite update subroutine.

Any structured programming language could be categorized as a message-based language. Message-based programming is a concept, a way to handle user input and control flow, a methodology rather than a type of language. It became the dominant feature of modern operating systems evolved from the command line to graphical user interfaces.

In a GUI based operating system, such as Windows, the OS manages the user interface. If the programmer isn't building a text edit field from scratch, he/she is just building a shell, there had to be a way to notify the programmer that the user wants to edit the text. The method is to send a message to the program indicating that the edit field needs to be updated. The borrowing of GUI elements and receiving of messages has been formalized in the Windows Software Development Kit, or more commonly, the Windows SDK.

The Windows SDK is a collection of application programming interfaces (APIs) that form the majority of the operating system. Any GUI based program uses the Windows SDK, even if it isn't readily apparent. In Rapid Application Development

Basic or Real Basic, the languages hide the details of the SDK by using they are using the SDK.

While RAD languages enable the programmer to quickly build GUI-based details of the SDK are not accessible. For example, it is quite difficult to straightforward using the SDK. However, the SDK is huge, and the shea programmers give up on the idea of SDK programming. The common th to use, but the opposite is true. Because the operating system handles a the programmer can concentrate on the most important aspect of progra program is all about user interaction.

FreeBasic doesn't have a RAD system for Windows programming. To cr will have to use the SDK, as this is the only option. While the SDK is ma fully understand, for 99% of all Windows programs, only a small subset that Windows SDK programming is no harder than any other type of pro actually easier than a language where you would have to create all the C

Putting aside all the gritty details of the Windows API for the moment, it of messages in an SDK program. This is best accomplished by looking a In the examples\Windows\gui folder of the FreeBasic .15b distribution (w there is a nice Hello World program that I am going to steal--I mean borr

```
Option Explicit
Option Private

#include once "windows.bi"

Declare Function WinMain ( ByVal hInstance
                          ByVal hPrevInst
                          szCmdLine As S
                          ByVal iCmdShow

''
'' Entry point
''
End WinMain( GetModuleHandle( null ), null, Comr
```

```

'' :::::::::::
'' name: WndProc
'' desc: Processes windows messages
''
'' :::::::::::
Function WndProc ( ByVal hWnd As HWND, _
                  ByVal message As UINT, _
                  ByVal wParam As WPARAM, _
                  ByVal lParam As LPARAM ) As LRESULT

    Function = 0

    ''
    '' Process messages
    ''
    Select Case( message )
        ''
        '' Window was created
        ''
        Case WM_CREATE
            Exit Function

        '' User clicked the form
        Case WM_LBUTTONDOWN
            MsgBox NULL, "Hello world from FreeB
        ''
        '' Windows is being repainted
        ''
        Case WM_PAINT
            Dim rct As RECT
            Dim pnt As PAINTSTRUCT
            Dim hDC As HDC

            hDC = BeginPaint( hWnd, @pnt )
            GetClientRect( hWnd, @rct )

            DrawText( hDC, _
                    "Hello Windows from FreeBasic
                    -1, _

```

```

        @rct, _
        DT_SINGLELINE Or DT_CENTER Or

    EndPaint( hWnd, @pnt )

    Exit Function

''
'' Key pressed
''
Case WM_KEYDOWN
    'Close if esc key pressed
    If( LoByte( wParam ) = 27 ) Then
        PostMessage( hWnd, WM_CLOSE, 0, 0 )
    End If

''
'' Window was closed
''
Case WM_DESTROY
    PostQuitMessage( 0 )
    Exit Function
End Select

''
'' Message doesn't concern us, send it to the de
'' and get result
''
Function = DefWindowProc( hWnd, message, wParam,

End Function

'' :~::~:
'' name: WinMain
'' desc: A win2 gui program entry point
''
'' :~::~:
Function WinMain ( ByVal hInstance As HINSTANCE, _
                  ByVal hPrevInstance As HINSTANCE,

```

```

        szCmdLine As String, _
        ByVal iCmdShow As Integer ) As Ir

Dim wMsg As MSG
Dim wcls As WNDCLASS
Dim szAppName As String
Dim hWnd As HWND

Function = 0

''
'' Setup window class
''
szAppName = "HelloWin"

With wcls
    .style          = CS_HREDRAW Or CS_VREDRAW
    .lpfnWndProc    = @WndProc
    .cbClsExtra     = 0
    .cbWndExtra     = 0
    .hInstance      = hInstance
    .hIcon          = LoadIcon( NULL, IDI_APPLICATION)
    .hCursor        = LoadCursor( NULL, IDC_ARROW)
    .hbrBackground = GetStockObject( WHITE_BRUSH)
    .lpszMenuName   = NULL
    .lpszClassName = StrPtr( szAppName )
End With

''
'' Register the window class
''
If( RegisterClass( @wcls ) = FALSE ) Then
    MessageBox( null, "Failed to register wcls!",
    Exit Function
End If

''
'' Create the window and show it
''

```

```

hWnd = CreateWindowEx( 0, _
                      szAppName, _
                      "The Hello Program", _
                      WS_OVERLAPPEDWINDOW, _
                      CW_USEDEFAULT, _
                      CW_USEDEFAULT, _
                      CW_USEDEFAULT, _
                      CW_USEDEFAULT, _
                      NULL, _
                      NULL, _
                      hInstance, _
                      NULL )

ShowWindow( hWnd, iCmdShow )
UpdateWindow( hWnd )

''
'' Process windows messages
''
While( GetMessage( @wMsg, NULL, 0, 0 ) <> FALSE
      TranslateMessage( @wMsg )
      DispatchMessage( @wMsg )
Wend

''
'' Program has ended
''
Function = wMsg.wParam

End Function

```

If you have successfully compiled and run the program, you will see a small form. If you click the form, a message box will be displayed, and if you click the close button, the form will close.

Take a moment to examine the window. You will see that the form has a menu and can be resized. Now look at the code above. There isn't any window properties, the OS handles all that for you. It also only takes a simple messagebox, which in itself, is a rather complex object. The ratio of results you were to try and recreate this simple program using FreeBasic's standard would be a hundred times larger.

The first thing you should notice about the code listed above is the form: a Windows program. Every Windows program, no matter how simple or complex. The two key ingredients of this program are the **WinMain** and **WinProc**

The WinMain procedure is the procedure Windows calls when a program starts a Windows program. In WinMain, you build and register the main program loop to process messages. Once the program enters the message loop, the WinProc procedure. Since this article is about the message model in a Windows message loop in WinMain and the WinProc procedure.

When the Windows operating system is running, there are messages being sent to a Windows program is running, the OS will send messages to the program that it thinks the program should know about. Some of these are program specific (or similar) procedure, and others, primarily user-generated messages, and most of a program is concerned with user interaction, it is important to understand

A queue is a data structure where data is added to the "back" of the queue and called a First-In-First-Out, or FIFO stack. If you have ever stood in line to get a queue.

For a program, the message queue will hold one or more messages, and the idle loop of a Windows program sits and waits for messages to arrive. The messages to the program. This message loop is contained within the following

```
''  
'' Process windows messages  
''  
While( GetMessage( @wMsg, NULL, 0, 0 ) <> FALSE  
    TranslateMessage( @wMsg )  
    DispatchMessage( @wMsg )
```

Wend

The **GetMessage** procedure retrieves a message from the queue via the `MSG` type-def that contains the necessary information related to a particular message. The `MSG` type-def.

```
Type MSG
  hwnd As HWND
  message As UINT
  wParam As WPARAM
  lParam As LPARAM
  Time As DWORD
  pt As Point
End Type
```

hwnd is the handle of the window that needs to process the message. This is the window's WinProc procedure.

Message is the message identifier. This could be, for example, `WM_CREATE` when the window has been created, but not yet shown.

wParam and **lParam** both specify additional information based on the message. For example, when a key is pressed, you can retrieve the key code by using the low byte of **wParam**.

time specifies the time that the message was posted and **pt** is a structure that contains the coordinates of the mouse when the message was posted.

TranslateMessage converts virtual key messages to character messages so that the key can be processed if desired. Any program that uses the key **DispatchMethod** then sends a message to the window's WinProc (or similar) window identified by the **hwnd** parameter.

To summarize the actions here, a user generated message will be placed in the message queue. GetMessage retrieves the first waiting message, passes it to TranslateMessage if necessary, and puts it back into the queue. The message is then passed to the message handler procedure to see which window should get the message, and then passed to the window handler procedure, which in our example, is WinProc.

Before we discuss the WinProc procedure however, we need to ask a question: How does WinMain know what procedure to use for a window? The answer is the **WNDCLASS** structure. In our example, **wcls** is defined as WNDCLASS

```
With wcls
    .style           = CS_HREDRAW Or CS_VREDRAW
    .lpfnWndProc    = @WndProc
    .cbClsExtra     = 0
    .cbWndExtra     = 0
    .hInstance      = hInstance
    .hIcon          = LoadIcon( NULL, IDI_APPLICATION)
    .hCursor        = LoadCursor( NULL, IDC_ARROW)
    .hbrBackground = GetStockObject( WHITE_BRUSH)
    .lpszMenuName   = NULL
    .lpszClassName = StrPtr( szAppName )
End With
```

As you can see, the WNDCLASS structure holds all of the information needed to register a window class. The important item is the **.lpfnWndProc** field. This field holds the address of the window procedure. The @ operator in FreeBasic returns the address of an object procedure. Once this window is registered using the **RegisterClass** method, the window procedure will be used to process messages.

As you can see, there is no special significance to the name WinProc. It is just a window handler. The actual SDK name is WindowProc, which is just a placeholder name. The important piece of information is that whatever you call the window procedure, it must have the same parameters as we have defined in our WinProc, and the address of the procedure must be assigned to the **.lpfnWndProc** field.

When you design a GUI program, you have to ask yourself, "How do I want to handle this?" For example, when the application is minimized, should the program ignore the event, or should it like put itself in the system tray? This is the essence of message-based programming: identifying important events, and then writing individual routines that handle each event. A collection of specific routines written in response to specific messages.

Despite the reputation of the SDK, the basic concept of message-based programming is writing a collection of routines to handle messages. This is the core task of GUI programming, or when to repaint the window is done by the operating system. It is the heart of the matter. There is a lot in there. However, like the cliché says, the best way to eat the SDK is to simply understand the concept of message-based programming: boilerplate code. Once that is done, creating sophisticated Windows programs is a matter of

NOTE! Have to do some spell checking, verify text, code and filenames.

Foreword

This is a tiny basic tutorial on how to write a simple library in C and then knowledge of C or FreeBASIC. After doing this tutorial you should be able to use the header files to FreeBASIC header files and understand how to use the li

What is a library

Prerequisite

This tutorial was written and tested with FreeBASIC 0.16b and the latest compiler tool chain. You also get code::blocks with a mingw32 bundle.

Formal description of the task at hand

To demonstrate usage of a C library in FreeBASIC we need to create the library works as intended. Then we have to translate the library header for the library.

Creating the files

So our file list will look like this:

myClib.c: C file implementing our library.

myClib.h: C header file describing the libraries interface.

myClibCTest.c: C file implementing our test program in C.

myClib.bi: FreeBASIC header file. A translation of myClib.h.

myClibFBTest.bas: FreeBASIC

make.cmd: A sample shell script compiling the library and test files.

The C file to become a static library. myClib.c

```

/* A function adding two integers and returning the result */
#include "myClib.h"
int SampleAddInt(int i1, int i2)
{
    return i1 + i2;
}

/* A function doing nothing ;) */
void SampleFunction1()
{
    /* insert code here */
}

/* A function always returning zero */
int SampleFunction2()
{
    /* insert code here */

    return 10;
}

```

The header file myClib.h

```

(C)
int SampleAddInt(int i1, int i2);
void SampleFunction1();
int SampleFunction2();

```

A C test project to verify that the static lib is C compatible. myClibC

```

(C)
#include
#include
#include "myClib.h"
int main(int argc, char *argv[])
{
    printf("SampleAddInt(5, 5):=%d\n", SampleAddInt(5, 5));
    system("PAUSE");
    return 0;
}

```

Translating the C header file to a FreeBASIC header file

myClib.bi: To interface the static library and automatically include it (#inc

```
'include file for libmyClib.a
#ifdef __myClib_bi__
#define __myClib_bi__
#include "myClib"

Declare Function SampleAddInt cdecl Alias "SampleAddInt"
Declare Sub SampleFunction1 cdecl Alias "SampleFunction1"
Declare Function SampleFunction2 cdecl Alias "SampleFunction2"
#endif
```

And finally the FreeBASIC file using the library

myClibFBTest.bas:

```
'Testing functions in myClib.bi
#include "myClib.bi"
''
Print "SampleAddInt(10, 10):=", SampleAddInt(10, 10)
' Just a dummy call
SampleFunction1()
''
Print "SampleFunction2():=", SampleFunction2()
```

The make file: make.cmd

I have created a batch file to compile all the files. Including a sample in (suite your setup.

```
(cmd)
@REM TODO: Set PATH's for this session.
SET PATH=C:\mingw32\bin;c:\mingw32\mingw32\bin
SET MINGW_INCLUDE="C:/MinGW32/include"
SET MINGW_LIB="C:/MinGW32/lib"
```

```

@REM
@REM fbc testing SET fbc="C:\portableapps\FreeBASIC\fbc.exe"
SET fbc="C:\FreeBasic16b\fbc.exe"
@echo *** Verify pat's to compilers
@pause
@echo off

@REM
@REM Remove old files
DEL /F *.o *.a myClibFBTest.exe

@REM
@REM Create static lib from c source
gcc.exe -c myClib.c -o myClib.o -I%MINGW_INCLUDE%

@REM
@REM ar: creating libstatictest.a
ar r libmyClib.a myClib.o

@REM
@REM No need for ranlib anymore? ar is supposed to take care of :
ranlib libmyClib.a

@REM
@REM Create a test with a C file

gcc.exe -c myClibCTest.c -o myClibCTest.o -I%MINGW_INCLUDE%
gcc.exe myClibCTest.o -o "myClibCTest.exe" -L%MINGW_LIB% libmyCl:

echo =====
echo RUnning C sample
echo =====
myClibCTest.exe

echo =====
echo Creating FreeBASIC sample
echo =====
REM I thought this explicit reference is unnecessary as I use #i
SET fbcop= -I myClib
SET fbcfl="myClibFBTest.bas"
%fbc% %fbcop% %fbcfl%
echo =====
echo RUnning FreeBASIC sample
echo =====
myClibFBTest.exe
@pause

```

Encountered error messages and their solutions

undefined reference to

Trying to link against the static C library without using the cdecl alias "fui

```
(cmd)
C:\code>"C:\FreeBasic16b\fb.exe"      "myClibFBTest.bas"
myClibFBTest.o:fake:(.text+0x3d): undefined reference to `SAMPLE/
myClibFBTest.o:fake:(.text+0x4a): undefined reference to `SAMPLE/
myClibFBTest.o:fake:(.text+0x67): undefined reference to `SAMPLE/
Press any key to continue . . .
```

To resolve this you will have to locate function declarations in a *.bi file th

```
Declare Function SampleAddInt(ByVal i1 As Integer, f
```

And change it to something like this:

```
Declare Function SampleAddInt cdecl Alias "SampleAdd
```

Appendix A: links

The basis for this tutorial is several threads in the forum.

When it evolves and can stand alone the links to the threads might be re
Some interesting links containing information on interfacing libraries cre:

[How do I compile a C project as a static lib for inclusion..](#)

SDL_Net: Getting Started



A complete Step by step guide of getting your program from hello world to hello world over a TCP/IP connection, using the SDL_Net SDL library. This tutorial will list all componets required and where to download them at the time of writing followed by how to get each componet in the prope place to perform the proper functions and finally how to write the actual code. I will assume you have zero previous knowledge and because of that some readers may want to skip the first few bits of the tutorial.

Written by GregF (Paragon)

Step 1: What you need.

Ok, lets pretend that you just sat down and installed the compiler and ar IDE. This list takes it from there.

- SDL_Net.bi - Installed with the compiller.
- SDL_Net.dll Binary -http://www.libsdl.org/projects/SDL_ne
- SDL.dll runtime library - <http://www.libsdl.org/download-1.2.php>

Step 2: Where you put it.

The .bi file can be put pretty much where ever you want to put it, you will tell the compiler where to find it in the **'\$Include** command. The .dll however need to be placed in specific places. The easiest way to make sure that the program will be able to use these files is to have the .dll in the same folder as the compiled executable. You can also put them in any folder that is listed in your Enviroment variable, but I don't recommend that because it will be easier to find and remeber that you need the .dlls if you just put them in the same folder as the executable, which will proably be the same folder as your .bas file for the main program.

Tutorial in progress...

Using FreeBASIC Built Libraries with GCC



by Jeff Marshall

Shows how to create a static library with FreeBASIC and then call it from

- *Minimum fbc version tested is v0.18.2b*

This article shows Windows usage throughout, but application to FreeB/

In this tutorial:

[A Simple Test](#)

[FreeBASIC Library With Dependencies](#)

[Using FreeBASIC as a Smart Linker](#)

A Simple Test

For this simple test we are going to create a FreeBASIC static library, or around, and will allow us to check that the basics are working:

First we need a library, and for for this it will be just a single trivial function the use of `cdecl` and `Alias` in our procedure definition. By default, C use declaration makes matching case sensitivity between FreeBASIC and C

```
' mylib1.bas

Function Add2Numbers cdecl Alias "Add2Numbers" _
    ( _
        ByVal x As Integer, _
        ByVal y As Integer _
    ) As Integer

    Return x + y

End Function
```

Create a file called `mylib1.bas` as above and compile it with:

```
fbc -lib mylib1.bas.
```

This will create our static library `libmylib1.a`. Next we need a C program prototype that exactly matches the function we have in the FreeBASIC library couple of variables to call `Add2Numbers()`, and print the results.

```
/* test1.c */  
  
#include <stdio.h>  
  
/* Prototype from libmylib.a */  
Int Add2Numbers( Int x, Int y );  
  
Int main ()  
{  
    Int a = 5;  
    Int b = 7;  
    Int c = Add2Numbers( a, b );  
  
    printf( "a = %d\n", a );  
    printf( "c = %d\n", b );  
    printf( "a + b = %d\n", c );  
  
    Return 0;  
}
```

To compile this C program using the FreeBASIC library we just made we need to specify which libraries are needed. In our case, it is `libmylib1.a`.

```
gcc test1.c -L . -l mylib1 -o test1.exe
```

The `'-L .'` option tells the linker to search in the current directory for the library we just created. This is the simplest case because the `libmylib1.a` library

for example the FreeBASIC run-time library libfb.a, we would need to sp

FreeBASIC Library With Dependencies

Here we create a FreeBASIC library that uses some features from the F to specify any additional needed libraries to GCC.

```
' ' mylib2.bas

Sub TestGfx cdecl Alias "TestGfx" ()

    Screen 12

    Line (0,0)-(100,100),15

    Sleep

End Sub
```

Create a file called mylib2.bas with the listing above and compile it with:

```
fbcc -lib mylib2.bas.
```

This will create our static library libmylib2.a. Next we need a C program prototype that exactly matches the function we have in the FreeBASIC li TestGfx() before terminating.

```
/* test2.c */

void TestGfx();

Int main()
{
```

```
    TestGfx();  
  
    Return 0;  
  
}
```

To compile and link `test2.c` directly with `gcc`, not only do we need to tell the linker that `libmylib2.a` needs.

```
gcc test2.c -L. -lmylib2 -L"C:\FreeBASIC\lib\win32" "C:\FreeBASIC\lib\win32\fb
```

Depending on what our FreeBASIC library uses, it we may use several options on the `gcc` command line. In this example, FreeBASIC is located in "C:\FreeBASIC\lib\win32". "C:\FreeBASIC\lib\win32\fb" is a special static library installed with FreeBASIC. "C:\FreeBASIC\lib\win32\fb" is a special static library specifically, it is initialized after the C runtime library, but before any of our other libraries. The actual libraries used, and which platform, for DOS or Linux, the program is being compiled for.

Using FreeBASIC as a Smart Linker

FreeBASIC has a neat built-in feature that stores a little bit of extra information about the libraries used, and which dependent libraries are needed. This is a FreeBASIC option that allows us to use `gcc` as the main compiler and linker.

If we reuse the examples from the previous section, `mylib2.bas` and `test2.c` can save ourselves a bunch of typing. Plus we usually won't have to know the exact path to the libraries. Compile `mylib2.bas` as before into a static library.

```
fbcc -lib mytest2.bas
```

Next we compile our C test program. Notice the `-c` option for the `gcc` compiler, which tells it to compile the source, but not link it yet. `test2.o` will still have the entry point, but we are not yet executable right away.

```
gcc -c test2.c -o test2.o
```

Lastly, we use `fbcc` to perform the link step. We are not compiling any basic capabilities of FreeBASIC such that the command line is fairly simple:

```
fbcc test2.o -l mylib2
```

This will create an executable named `test2.exe` because `test2.o` was supplied with information stored in `libmylib2.a` and automatically know which additional libraries are needed, especially when many extra FreeBASIC built libraries are needed.

See also

- **Static Libraries**

Written by rdc

Introduction

FreeBASIC is moving towards implementing Object Oriented programming added to the language, the Type definition has been extended to include first step towards full class support. This article introduces some of the c explains some of the extended type constructs.

Object Oriented Programming

Object Oriented Programming, usually shortened to OOP, is a methodology to build code units called objects. An object is a thing; it is a unit of code that can be manipulated in a program. You can think of an object as a noun: a person, a sprite, a drawing primitive or something more elaborate like a tank in a game. A set of characteristics and actions can be represented as an object.

An object contains both the data needed by the object, and the methods that operate on the data. This grouping of data and methods into a single entity is called encapsulation. This allows you to create modular units that can be reused in multiple programs. This is the motivation in the creation of the OOP paradigm.

Another beneficial consequence of encapsulation is information hiding. By hiding the data from the outside world so that unwanted changes to the data cannot occur directly, the object has a public interface that the external program should use. By using an interface, you can control how the object behaves and remain consistent across many programs.

The interface also allows you to make internal changes to the code, without being accessed. As long as you don't change the published interface, that is, the public interface, you can improve the object without breaking any existing code that relies on the old interface. If a program may need an improved method, you can leave the old method in place and just add a new method with the improved functionality. New programs can use the new method, while old programs can still use the old method.

Another advantage of using a public interface is so that other programmers can use the object without knowing its internal details.

worrying about the internal details of the object. As long as the published interface is documented, anyone should be able to use your object, even beginners

The Published Contract

As already stated, OOP was designed to enable code reuse among programmers. To be helpful, the published interface must remain stable. That is, once an object is used in programs, the published interface should not change so that programs will continue to work correctly. There is an implicit contract between you as the author of an object that you will maintain the published interface across changes that you make. This implicit contract between author and user is the main strength of OOP. One of the reasons that OOP has become such a powerful programming methodology

The Characteristics of an Object

As already mentioned, an object contains both data and methods. The data members describe what the object contains, while the methods describe what the object can do. A simple example will illustrate this concept.

Suppose you want to create an object that draws a rectangle on the screen. The data members that would be contained within the data members of the object are the x and y coordinates of the top left corner, the width and a height, so the object would have width and height data members. To draw the rectangle outlined or filled, so a filled flag data member can be added to the object. To draw the rectangle in a particular color, so the object will need a color data member. To have the object be a bit more flexible, you can add a color member for the fill. Of course you will need a method to actually draw the rectangle. So you will add a draw routine to the object definition.

So our rectangle object has the following preliminary properties and methods:

Property: x and y origin

Property: width

Property: height

Property: filled

Property: outline color

Property: fill color

Method: DrawRect

This list is called the object definition. In FreeBASIC you define an object. The extended Type is similar to the standard Type, with some added language subset of OOP features.

A Rectangle Type Definition

The following code snippet is a partial rectangle definition:

```
Type myRect
  Private:
    X_ As Integer
    Y_ As Integer
    Width_ As Integer
    Height_ As Integer
    Filled_ As Integer
    Outlinecolor_ As Integer
    Fillcolor_ As Integer
  Public:
    Declare Sub DrawRect()
End Type
```

As you can see, the extended Type looks much like a standard Type except for the `Private:` keyword and the sub declaration. The `Private:` keyword tells the compiler that the members listed below are private to the type, that is cannot be accessed outside of the type. The compiler will hide these members until a new qualifier is encountered, which in this case is the `Public:` declaration. All of the data members are hidden from the outside world and are only accessible within the scope of the Type, a process called information hiding. The underscore is the common way to define private variables.

Information hiding is a way to maintain the integrity of the object. You should not directly access a data member. All data access should be through the use of methods. This way you can control what is being passed to your object. Strict control over your object helps prevent errors that may occur when a programmer uses your object.

```
Type myRect
  Private:
```

```

X_ As Integer
Y_ As Integer
Width_ As Integer
Height_ As Integer
Filled_ As Integer
Otlncolor_ As Integer
FillColor_ As Integer
Public:
Declare Sub DrawRect()
Declare Property X(ByVal xx_ As Integer)
Declare Property X() As Integer
Declare Property Y(ByVal yy_ As Integer)
Declare Property Y() As Integer
Declare Property Width(ByVal w_ As Integer)
Declare Property Width() As Integer
Declare Property Height(ByVal h_ As Integer)
Declare Property Height() As Integer
Declare Property Filled(ByVal f_ As Integer)
Declare Property Filled() As Integer
Declare Property Otlncolor(ByVal oc_ As Integer)
Declare Property Otlncolor() As Integer
Declare Property FillColor(ByVal fc_ As Integer)
Declare Property FillColor() As Integer
End Type

```

The Declare statements following the Public: qualifier comprises the public variables of the type are defined with the Private: keyword, the only way Property members maintaining the integrity of the object. Since you define you have full control over what is being put into your object. A common code in your property members so that the object does not contain invalid

In this example, the variables can be both written and read. The compile Property and a write Property by the type of the method. A subroutine-formatted since you are passing a value that will be saved in a private variable. A function property since a private variable will be returned to the caller. You can create just a function-formatted Property or write-only Properties by just adding

Creating Well-Behaved Objects

The definition looks complete at this point, but there is a problem. What variables were not initialized? The object would not perform correctly and it would be better to have a set of default values for the object variables just to get initialized. You can initialize the object at the moment of creation by using a constructor.

A Constructor is a subroutine that is called when the object is created using Constructors are useful for initializing an object, either with default values or with specific values. The updated type definition now looks like the following:

```
Type myRect
Private:
    X_ As Integer
    Y_ As Integer
    Width_ As Integer
    Height_ As Integer
    Filled_ As Integer
    Outlinecolor_ As Integer
    Fillcolor_ As Integer
Public:
    Declare Sub DrawRect()
    Declare Property X(ByVal xx_ As Integer)
    Declare Property X() As Integer
    Declare Property Y(ByVal yy_ As Integer)
    Declare Property Y() As Integer
    Declare Property Width(ByVal w_ As Integer)
    Declare Property Width() As Integer
    Declare Property Height(ByVal h_ As Integer)
    Declare Property Height() As Integer
    Declare Property Filled(ByVal f_ As Integer)
    Declare Property Filled() As Integer
    Declare Property Outlinecolor(ByVal oc_ As Integer)
    Declare Property Outlinecolor() As Integer
    Declare Property FillColor(ByVal fc_ As Integer)
    Declare Property FillColor() As Integer
    Declare Constructor()
    Declare Constructor(xx_ As Integer, yy_ As Integer,
                       w_ As Integer, h_ As Integer, f_ As Integer,
```

```
fc_ As Integer )
```

```
End Type
```

You will notice in the definition that we have two Constructors, one that doesn't. This is called overloading and can be used not only with Constructors and functions. Overloading is useful for situations where you need to have a single method call. The compiler will determine which method to call based on the method. You can overload as many methods as you want, as long as each method is unique.

In this instance, if the Constructor is not passed any parameter values, it will use default values. If the Constructor is called with parameters, then it will use those values for the object's variables.

There is also a Destructor method that is called when the object is destroyed. It can perform any cleanup tasks that must be carried out before the object is removed. For example, if you created any pointer references, or opened any files, then you would clean up in the Destructor. Since the Rectangle object doesn't create any outside references, it doesn't need a Destructor.

Filling in the Object Methods

The type definition is a template for the object type and tells the compiler how to use it. However, in order to actually use the object, you need to create the actual object. The next listing shows how to do this.

```
Type myRect
  Private:
    X_ As Integer
    Y_ As Integer
    Width_ As Integer
    Height_ As Integer
    Filled_ As Integer
    Outlinecolor_ As Integer
    Fillcolor_ As Integer
  Public:
  Declare Sub DrawRect()
```

```

Declare Property X(ByVal xx_ As Integer)
Declare Property X() As Integer
Declare Property Y(ByVal yy_ As Integer)
Declare Property Y() As Integer
Declare Property Width(ByVal w_ As Integer)
Declare Property Width() As Integer
Declare Property Height(ByVal h_ As Integer)
Declare Property Height() As Integer
Declare Property Filled(ByVal f_ As Integer)
Declare Property Filled() As Integer
Declare Property OutlineColor(ByVal oc_ As Integer)
Declare Property OutlineColor() As Integer
Declare Property FillColor(ByVal fc_ As Integer)
Declare Property FillColor() As Integer
Declare Constructor()
Declare Constructor(xx_ As Integer, yy_ As Integer,
                   h_ As Integer, f_ As Integer,
                   fc_ As Integer)

End Type

Sub myRect.DrawRect()
    Line (this.x_, this.y_)-
    (this.x_ + Width - 1, this.y_ + this.height_ - 1), t
    If this.Filled_ <> 0 Then
        Paint (this.x_ + 1, this.y_ + 1), this.FillCo
    End If
End Sub

Property myRect.x(ByVal xx_ As Integer)
    this.X_ = xx_
End Property

Property myRect.x() As Integer
    Return this.X_
End Property

Property myRect.y(ByVal yy_ As Integer)
    this.Y_ = yy_
End Property

```

```
Property myRect.y() As Integer
    Return this.y_
End Property
```

```
Property myRect.Width(ByVal w_ As Integer)
    this.Width_ = w_
End Property
```

```
Property myRect.Width() As Integer
    Return this.Width_
End Property
```

```
Property myRect.Height(ByVal h_ As Integer)
    this.Height_ = h_
End Property
```

```
Property myRect.Height() As Integer
    Return this.Height_
End Property
```

```
Property myRect.Filled(ByVal f_ As Integer)
    this.Filled_ = f_
End Property
```

```
Property myRect.Filled() As Integer
    Return this.Filled_
End Property
```

```
Property myRect.OutIncolor(ByVal oc_ As Integer)
    this.OutIncolor_ = oc_
End Property
```

```
Property myRect.OutIncolor() As Integer
    Return this.OutIncolor_
End Property
```

```
Property myRect.FillColor(ByVal fc_ As Integer)
    this.Fillcolor_ = fc_
End Property
```

```
End Property
```

```
Property myRect.FillColor() As Integer  
    Return this.Fillcolor_  
End Property
```

```
Constructor myRect  
    this.X_ = 0  
    this.Y_ = 0  
    this.Width_ = 10  
    this.Height_ = 10  
    this.Filled_ = 0  
    this.OutIncolor_ = 15  
    this.Fillcolor_ = 7  
End Constructor
```

```
Constructor MyRect (xx_ As Integer, yy_ As Integer,  
                    h_ As Integer, f_ As Integer,  
                    fc_ As Integer)  
  
    this.X_ = xx_  
    this.Y_ = yy_  
    this.Width_ = w_  
    this.Height_ = h_  
    this.Filled_ = f_  
    this.OutIncolor_ = oc_  
    this.Fillcolor_ = fc_  
End Constructor
```

The Methods and Properties are defined using the Sub/Function/Property. This tells the compiler how to match up methods with the proper type definition with the type name for the same reason. The *this* identifier is a hidden property that refers to the defined type. You use the *this* identifier to specify that y

Using Your Object

The object is now complete can be used in a program which is listed below

```

Type myRect
  Private:
    X_ As Integer
    Y_ As Integer
    Width_ As Integer
    Height_ As Integer
    Filled_ As Integer
    Otlncolor_ As Integer
    Fillcolor_ As Integer
  Public:
    Declare Sub DrawRect()
    Declare Property X(ByVal xx_ As Integer)
    Declare Property X() As Integer
    Declare Property Y(ByVal yy_ As Integer)
    Declare Property Y() As Integer
    Declare Property Width(ByVal w_ As Integer)
    Declare Property Width() As Integer
    Declare Property Height(ByVal h_ As Integer)
    Declare Property Height() As Integer
    Declare Property Filled(ByVal f_ As Integer)
    Declare Property Filled() As Integer
    Declare Property Otlncolor(ByVal oc_ As Integer)
    Declare Property Otlncolor() As Integer
    Declare Property FillColor(ByVal fc_ As Integer)
    Declare Property FillColor() As Integer
    Declare Constructor()
    Declare Constructor(xx_ As Integer, yy_ As Integer,
                       h_ As Integer, f_ As Integer,
                       fc_ As Integer)

End Type

Sub myRect.DrawRect()
  Line (this.x_, this.y_)-
  (this.x_ + this.Width_ - 1, this.y_ + this.height_ - 1)
  If this.Filled_ <> 0 Then
    Paint (this.x_ + 1, this.y_ + 1), this.FillColor
  End If
End Sub

```

```
Property myRect.x(ByVal xx_ As Integer)
    this.X_ = xx_
End Property
```

```
Property myRect.x() As Integer
    Return this.X_
End Property
```

```
Property myRect.y(ByVal yy_ As Integer)
    this.Y_ = yy_
End Property
```

```
Property myRect.y() As Integer
    Return this.y_
End Property
```

```
Property myRect.Width(ByVal w_ As Integer)
    this.Width_ = w_
End Property
```

```
Property myRect.Width() As Integer
    Return this.Width_
End Property
```

```
Property myRect.Height(ByVal h_ As Integer)
    this.Height_ = h_
End Property
```

```
Property myRect.Height() As Integer
    Return this.Height_
End Property
```

```
Property myRect.Filled(ByVal f_ As Integer)
    this.Filled_ = f_
End Property
```

```
Property myRect.Filled() As Integer
    Return this.Filled_
End Property
```

```
Property myRect.Otlncolor(ByVal oc_ As Integer)
    this.Otlncolor_ = oc_
End Property
```

```
Property myRect.Otlncolor() As Integer
    Return this.Otlncolor_
End Property
```

```
Property myRect.FillColor(ByVal fc_ As Integer)
    this.Fillcolor_ = fc_
End Property
```

```
Property myRect.FillColor() As Integer
    Return this.Fillcolor_
End Property
```

```
Constructor myRect
    this.X_ = 0
    this.Y_ = 0
    this.Width_ = 10
    this.Height_ = 10
    this.Filled_ = 0
    this.Otlncolor_ = 15
    this.Fillcolor_ = 7
End Constructor
```

```
Constructor MyRect (xx_ As Integer, yy_ As Integer,
                    h_ As Integer, f_ As Integer,
                    fc_ As Integer)
```

```
    this.X_ = xx_
    this.Y_ = yy_
    this.Width_ = w_
    this.Height_ = h_
    this.Filled_ = f_
    this.Otlncolor_ = oc_
    this.Fillcolor_ = fc_
End Constructor
```

```

'Create a graphic screen
Screen 18

'Create an object using the default constructor
Dim aRect As myRect
'Create an object by explicitly setting the constructor
Dim bRect As myRect = myRect(200, 200, 200, 100, 1,

'Draw the rectangles on the screen
aRect.DrawRect
bRect.DrawRect

'Update aRect properties
aRect.X = 90
aRect.Y = 20
aRect.Filled = 1
aRect.FillColor = 15

'Draw new rect
aRect.DrawRect
Sleep
End

```

To initialize the object using the default Constructor, you simply Dim the standard type. If the Constructor only takes a single value then you can use the standard syntax. To initialize the object with a set of values, you use the Dim type and parm1...) syntax. You can see that accessing the members of the object is done using the standard type.

Thanks to cha0s at the FreeBASIC forums for the information regarding

Simulating Polymorphism



Written by *rdc*

Introduction

Polymorphism is a powerful tool in object-oriented program. A polymorphic function behaves differently depending on the definition of the object. For example, an animal object may have a speak method that will issue a bark for a dog. FreeBasic doesn't support true polymorphism before version 0.90.0. How to simulate polymorphic methods using method pointers.

Polymorphism

Polymorphic methods are subroutines or functions that have the same type list, but behave differently when bound to different objects. An animal object may have a speak method that will issue a bark for a dog and a meow for a cat. Since FreeBasic does not yet have classes, you cannot implement true polymorphic methods, but you can simulate the behavior by using method pointers.

The following listing shows a couple of defines and an extended type definition.

```
#define isdog 1
#define iscat 2

Type animal
  Public:
    speak As Sub()
    Declare Constructor (anid As Integer)
End Type
```

The #defines are passed to the Constructor to signal what type of object you are creating. The speak As Sub() definition defines the method pointer. As you will see, the different subroutines will be passed to the speak method pointer. The following listing shows the different speak subroutines and the Constructor method:

```
'Speak method for dog object
Sub Bark()
```

```

        Print "Woof!"
    End Sub

    'Speak method for cat object
    Sub Meow()
        Print "Meow!"
    End Sub

    'Set the proper method pointer based on animal id
    Constructor animal(anid As Integer)
        If anid = isdog Then
            this.speak = @Bark
        ElseIf anid = iscat Then
            this.speak = @Meow
        End If
    End Constructor

```

The Bark subroutine will be called if the object is a dog and the Meow sub called if the object is a cat. You may be wondering why you can't just overload. For overloaded methods, the type and parameter list must be unique, with the same name. Since Bark and Meow have the same name and parameter list, that is no parameters, you cannot overload the method.

The Constructor code is where the program decides what method call to use. If anid is equal to isdog, then the Speak method pointer will be set to the address of the Bark method. If anid is equal to iscat then Speak will be set to the address of the Meow method. The addressof operator @ is used to pass the address of Bark and Meow to the Constructor.

The *this* object reference is a hidden parameter that is passed to the Constructor. It references the type, which in this case is animal. You can use this to reference variables within the type.

The only thing left to do is to create and initialize the object:

```

'Create a dog and cat object
Dim myDog As animal = isdog
Dim mycat As animal = iscat

```

Here myDog and myCat are created with the appropriate flags passed to them so that the proper references can be set up. Once the objects are created you call the speak method of each object.

```
'Have the animals speak
Print "My dog says ";
myDog.speak()
Print "My cat says ";
myCat.speak()
```

Notice that you are calling the same speak method, yet the output is different.

```
My dog says Woof!
My cat says Meow!
```

This is the essence of polymorphic methods.

Here is the complete program listing:

```
'Simulated Polymorphism Using Method Pointers
'Richard D. Clark
'Requires the CVS version of FreeBasic
|*****

#define isdog 1
#define iscat 2

Type animal
  Public:
    speak As Sub()
    Declare Constructor (anid As Integer)
End Type

'Speak method for dog object
Sub Bark()
  Print "Woof!"
```

```

End Sub

'Speak method for cat object
Sub Meow()
    Print "Meow!"
End Sub

'Set the proper method pointer based on animal id
Constructor animal(anid As Integer)
    If anid = isdog Then
        this.speak = @Bark
    ElseIf anid = iscat Then
        this.speak = @Meow
    End If
End Constructor

'Create a dog and cat object
Dim myDog As animal = isdog
Dim mycat As animal = iscat

'Have the animals speak
Print "My dog says ";
myDog.speak()
Print "My cat says ";
myCat.speak()

Sleep
End

```

From fbc version 0.90.0, polymorphism through inheritance and vir

Previous example transposed for fbc version 0.90.0 or greater, by using through inheritance with abstract/virtual methods (feature now supported)

```
'Requires FreeBasic version >= 0.90.0
```

```

'Base-type animal
Type animal Extends Object
    Declare Abstract Sub speak ()
End Type

'Derived-type dog
Type dog Extends animal
    Declare Virtual Sub speak () Override
End Type

'Speak method for dog object
Virtual Sub dog.speak ()
    Print "Woof!"
End Sub

'Derived-type cat
Type cat Extends animal
    Declare Virtual Sub speak () Override
End Type

'Speak method for cat object
Virtual Sub cat.speak ()
    Print "Meow!"
End Sub

'Create a dog and cat as dynamic object through anir
Dim myDog As animal Ptr = New dog
Dim mycat As animal Ptr = New cat

'Have the animals speak
Print "My dog says ";
myDog->speak()
Print "My cat says ";
myCat->speak()

Sleep

>Delete the dynamic objects

```

```
Delete myDog  
Delete myCat
```

OOP In Non-OOP Languages



Contrary to popular belief object oriented programming does not require

What you get with an OO language is a set of built in constructs that in many cases they are unnecessary and sometimes they are counterproductive.

Anyway, this isn't a rant against OO languages but rather a rant against a specifically OO language is necessary to write object oriented programs.

In order to demonstrate that it is not necessary to have an OO language usually presented as an example of class based programming; and so it is an example.

The code was tested using FB 0.16 for win32.

If you have to concatenate a lot of strings in most Basics you usually find FreeBasic string operations are remarkably quick but you can still do better.

A string builder is simply a class that maintains a string buffer in such a way that allocation function because this is a relatively expensive operation. The cost of manipulating the buffer and converting between it and the native string type is high.

The trick that makes it faster than the built type for large strings and large buffers is in a heap allocated buffer that is always larger than the actual length of the string. The end of the string usually simply means copying the contents of the new string to the end of the current string. In this implementation the buffer is a ZString dynamic string.

The FreeBasic module encapsulates a type definition for a struct. Instances of the object. The methods are simply normal FreeBasic public functions and so to want to call a method you use the normal FreeBasic syntax:

```
s = StringB_ToString(AStringBInstance)
```

By convention all methods names begin with the name of the class and always the instance of the type. This argument should always be passed.

state are permanent and also to avoid unnecessary, time-consuming, co

To add a new method you simply add a new function or sub following the

You can easily implement composition of objects but inheritance in the u
extend classes simply by defining new functions elsewhere that take arg
defines all of its methods as overloaded you can even create new metho
different signatures.

Here is the example code:

```
' -----  
' Classes without built in oop.  
  
' Define a struct for the properties and a sub or fu  
' method. Pass the struct as the first argument in  
  
' By convention the argument will be Me as in VB Cla  
  
' Strings in FB are so fast that a string builder c  
' not needed most of the time but if you are concate  
' thousands of strings to build web pages for instan  
  
' And please don't start complaining about the lack  
' is not a requirement for the use of objects. Ther  
' Object Oriented Programming but the most important  
' is the close association between the data and the  
  
'You can easily extend this class to provide more me  
' -----
```

Type StringB

```
    Len As Integer ' used length  
    allocated As Integer  
    s As ZString Ptr ' buffer of at least len charac  
End Type
```

```

' -----
' Create a new StringB by calling one of these constructors
' -----
Public Function StringB_New Overload (ByVal InitialSize As Integer) As StringB
    Dim sb As StringB
    sb.allocated = InitialSize
    sb.s = Allocate(InitialSize)
    *sb.s = ""
    StringB_New = sb
End Function

Public Function StringB_New(ByRef InitialValue As String) As StringB
    Dim sb As StringB
    sb = StringB_New(Len(InitialValue))
    *sb.s = InitialValue
    sb.len = Len(InitialValue)
    StringB_New = sb
End Function

Public Sub StringB_Dispose(ByRef Me As StringB)
    Deallocate Me.s
End Sub

Public Function StringB_ToString(ByRef Me As StringB) As String
    StringB_ToString = *Me.s
End Function

Sub StringB_Append Overload(ByRef Me As StringB, ByRef s As String)

    Dim i As Integer = Me.len
    Me.len += Len(s)
    If Me.len >= Me.allocated Then
        Me.allocated = 2*Me.len
        Dim As ZString Ptr p = Reallocate(Me.s, Me.allocated)
        If p=0 Then

```

```
        ' failed to reallocate
        Print "StringB_Append failed to reallocate", M
        Return
    End If
    Me.s = p
End If
*(Me.s + i) = s

End Sub

Sub StringB_Append(ByRef Me As StringB, ByRef other
    StringB_Append Me, StringB_ToString(other)
End Sub
```

Const Qualifiers and You



Note: As with all things regarding scope, Const qualifiers may be a bit difficult attempting to understand Const qualifiers.

Also note my cliché title, which I chose because of its cliché nature.

What the heck are Const qualifiers? Const qualifiers are a feature recent FreeBasic too. Const qualifiers are yet another form of protection - they allow some parts of the program to access (read) them but not modify them. They are very useful in OO situations, but you can probably benefit from them in other ways.

The Const qualifier in FreeBasic is essentially an extension to data types. Generally you put it right after the "As" part of the variable's data type declaration.

```
Dim As Const Integer my_const_int = 5
```

(By the way, throughout this tutorial I use only Integers and Integer Ptr types, including Types, Enums, and anything else that declares something.

Note in this case we are allowed to change it once - when we create it. FreeBasic will give an error if you don't (interestingly, you are allowed to set it equal to anything that modifies it after that. It will actually give you an error if, for

```
my_const_int = 3
```

Yet, since this doesn't change the variable any, you can do

```
Print my_const_int
```

Now this is all very good, but it doesn't seem much different from the normal

purposes, the same thing:

```
Dim As Const Integer my_const_int = 5
Const my_int As Integer = 5
```

Do they? Not quite. You see, the Const qualifier allows you to create constants inside Types and other places. What's more, you can put them inside Sub

```
Sub my_sub (some_num As Integer)
End Sub
```

Normally functions are allowed to modify the variables you send to them depends on whether you use ByVal or ByRef (and of course pointers is also may be undesirable, for whatever reason, and the Const qualifier exists. Normally it would only be a local copy that is modified, which is fine, since

```
Sub my_sub (ByRef some_num As Integer)
End Sub
```

Now my_sub has direct access to whatever variable you pass to it, and

```
my_sub(my_const_int)
```

Why? Simply because the function may modify the variable. We don't know you try to compile that is "Invalid assignment/conversion." It's almost as if it act like trying to pass a string to an integer argument (or vice-versa). Yet it possibly modify the variable!

And of course, if we did something like this:

```
Sub my_sub (ByRef some_num As Const Integer)
End Sub
```

Then it compiles just fine, but if you try to do the following within the function:

```
some_num = 3
```

Why? Once again, the original variable has been passed ByRef to the sub, but you cannot modify the original, which cannot be done. Once again, it's entirely possible to work around this:

```
Dim As Integer copy_of_some_num = some_num
copy_of_some_num = 3
```

But you can't modify some_num itself!

Now we come to pointers. What about them? For pointers it's a bit more complex - or even BOTH! So all of the following are valid:

```
Declare Sub my_sub_a (ByRef ptr_A As Const Byte Ptr)
Declare Sub my_sub_b (ByRef ptr_B As Byte Const Ptr)
Declare Sub my_sub_c (ByRef ptr_C As Const Byte Const Ptr)
```

The first one makes it so you can change the pointer itself all you want, the second allows you to change what the pointer points to, but you cannot change the pointer itself! In all cases you can make a copy of the pointer - *but it does not change the contents of whatever the original pointer points to!* This is

In case the behaviour of the Const qualifier seems a bit strange to you, I summed up pretty quickly: The Const qualifier aims to protect the original data. Remembering this will help you (there's pointers involved there are so many different places to put the Const there are!) So long as you remember what the Const qualifier is for, you need to *not* use it).

You can also use the Const qualifier in UDTs. In fact, it's actually a very interesting (and OOP nevertheless are very much related) - but even if you don't use OOP as an example, as it's pretty obvious by now how it works, but here's an example:

```
Type my_type
  As Const Integer t_int= 5
End Type

Dim As my_type t

t.t_int = 3
```

And obviously this won't compile, since the member t_int is Const. Furthermore, the following will not compile either, since ALL members of t are Const:

```
Type my_type
  As Integer t_int= 5
End Type

Dim As Const my_type t

t.t_int = 3
```

As for the OOP side of things (and if you aren't interested in OOP you can use ByVal as this when called. Is there a way to create constant objects? Of course not. Is there a distinction? The answer is yes. As of November 23, 200

```

Type my_object
  Public:
    Declare Sub modifier_sub ()

    'Subs that do not modify the object are declared
    Declare Const Sub non_modifier_sub ()
  Private:
    some_num As Integer = 3
End Type

Sub my_object.modifier_sub ()
  this.some_num = 3
End Sub

Sub my_object.non_modifier_sub()
  Print this.some_num
End Sub

'Note that only Const objects must be initialized (t
'just like variables. Thus, you must either have a
'default initial values (as I did here), in which ca
Dim As Const my_object t = my_object
Dim As my_object u

'Both of these will compile:
t.non_modifier_sub()
u.non_modifier_sub()

'...but the first of these will not compile, since r
t.modifier_sub()
u.modifier_sub()

'Sleep so we can see the results
Sleep

```


<http://sourceforge.net/tracker/index.php?func=detail&aid;=1480621&gro>

Creating and understanding your FBgfx image and font buffers

The FBgfx Image Buffer

Creating Buffers

Buffer Format

Getting Pixels

The FBgfx Font Header

Header Details

Creating a Font Buffer

Assigning Font Characters

Tips & Tricks

Coloring your Custom Fonts

ScrPtr vs ImgBuf

Download Accompanying Tutorial Files: [FreeBASIC Font Tutorial.7z](#)

The FBgfx Image Buffer

FBgfx has a new data type in .17 and above. This type is called `IMAGE`. You `#include "fbgfx.bi"` and then accessing the namespace for FBgfx, via `fb.Image` going to be using the `Ptr` type. A pointer, because it's dynamic

To use an image in the FBgfx Library, you have to create it via `image` buffer (made available) for your image. You have to deallocate (free, make available) it at the end of your program. FBgfx has its own internal pixel format, as created. The image header contains information about your image. This header contains the actual colors for each individual pixel in RGB (red, blue, green)

Creating Buffers

The size of the buffer you create will vary depending on screen depth. You need 4 bytes per individual pixels. Thus, a 32-bit pixel depth screen will need 4 bytes per pixel however, as using the `fb.Image` `Ptr` setup to create your buffer makes it easier. You only need to know this information to understand how much size a buffer

Actually creating the buffer is very simple. It's just a simple creation of a

```
#include "fbgfx.bi"

'' Our image width/height
Const ImgW = 64
Const ImgH = 64

'' Screens have to be created before a call to image
ScreenRes 640, 480, 32

'' Create our buffer
Dim As FB.Image Ptr myBuf = ImageCreate(ImgW, ImgH)

'' Print the address of our buffer.
Print "Buffer created at: " & myBuf
Sleep

'' Destroy our buffer. Always DESTROY buffers you
ImageDestroy( myBuf )
Print "Our buffer was destroyed."
Sleep
```

Code Dissection

```
#include "fbgfx.bi"
```

This includes the header file which contains the definition for the fb.Image

```
'' Our image width/height
Const ImgW = 64
Const ImgH = 64
```

This creates constants which will be used to decide the size of our image to `ImageCreate` when we use it.

```
' ' Screens have to be created before a call to image
ScreenRes 640, 480, 32
```

This creates our FBgfx screen. `ImageCreate` needs to know our bit depth parameter allowing you to set the depth yourself.

```
' ' Create our buffer
Dim As FB.Image Ptr myBuf = ImageCreate(ImgW, ImgH)
```

This first of all creates a pointer that is of the `fb.Image` type. It's just a local variable, right now it equals zero, and could not be used. That's considered the default value for a pointer.

The `ImageCreate` call returns the address of an area in memory of a new size of this buffer depends on the bit depth, but the width/height of the image is the same as the screen. `ImageCreate` can also take a fill color and depth as the third and fourth parameters. The buffer can be created filled with the transparent color and match the current screen's depth.

We now have allocated a space in memory. It's enough space to hold an image of its `fb.Image` type. We'll need to destroy it later for proper memory management.

```
' ' Print the address of our buffer.
Print "Buffer created at: " & myBuf
Sleep
```

This is just there to let you know what we've done. We print the address of the buffer to see if it worked.

```

    '' Destroy our buffer. Always DESTROY buffers you
ImageDestroy( myBuf )
Print "Our buffer was destroyed."
Sleep

```

Here we destroy our buffer with a call to `ImageDestroy`. We don't have to do this for consistency and clarity.

Buffer Format

Now that we know how to create buffers, we might want to know more in detail. We can open up the `fbgfx.bi` header file and find the `fb.Image` type, and you can see the format.

We actually don't need to know much about the format itself. The reason is that the `Buf + SizeOf(fb.Image)` in memory belongs to pixels. Everything before that is just a header because we used the `fb.Image Ptr`. All you have to know is what you want to store in the buffer.

FB.IMAGE Data Type

```

    '' Image buffer header, new style (incorporates old style)
Type IMAGE Field = 1
    Union
        old As _OLD_HEADER
        Type As UInteger
    End Union
    bpp As Integer
    Width As UInteger
    height As UInteger
    pitch As UInteger
    _reserved(1 To 12) As UByte
End Type

```

This same information can be found in `fbgfx.bi`. As you can see, this data is stored in a header.

The Width, Height, Pitch (bytes per row), and Bit Depth (bytes per pixel) and the old header itself within the same space. The new header format not used in the default dialect in the newer versions of FB, so we're not

How do we access that information within the header? If you're familiar with a buffer in the first example), then all you have to do is access your buffer and leave you to believe that all that's contained in your buffer is the `fb.Image` allows the compiler to think that's what's contained in the buffer, even though

Getting Pixels

The first section of our buffer which FreeBASIC helps us out with contains the address, and the rest of our buffer contains pixels (Example2.bas).

```
' We have to include this to use our FB.IMAGE data type
#include "fbgfx.bi"
```

Remember to include our `fb.Image` data type!

```
' This one is very important.
' We cast to a uByte ptr first off, to get the exact address
' We then cast to a uLong ptr, simply to avoid "signed integer"
' warnings.
Dim As uLong Ptr myPix = Cast( uLong Ptr, ( Cast( UByte Ptr, &fb.Image + sizeof(fb.Image) ) ) )
```

Phew. Alright. We have to make sure we get the exact address of our pixels. RGB, and the extra is generally used for alpha when you need it (some channels - to store all kinds of data). If we're even ONE BYTE off, your RGB we have to cast to a `UByte Ptr` first.

You probably also noticed that we simply added `sizeof(fb.Image)` to our buffer's size to the start of the buffer, we have just skipped all the memory address

Finally, we cast it all to a `uLong Ptr`, mainly for safety. We're in 32 bit dep

Here's a small line if you still don't understand how this works. Here is o

If what's contained in the first section of our buffer is the `fb.Image Header` our address for the pixels, simply by adding the size of the `fb.Image data`

One problem though! If we add that size to our buffer address, to try and because our datatype isn't one byte long. We have to cast to a `UByte Ptr` we'll get the exact byte we need in memory to work with.

Finally, we're in 32-bits. We just casted to a `UByte Ptr`. Although we *can practice to cast it to a `Ulong Ptr` first. We finally have the address of our manipulate those pixels directly now, if we'd like.

```
' ' Print information stored in our buffer.  
Print "Image Width: " & myBuf->Width  
Print "Image Height: " & myBuf->Height  
Print "Image Bit Depth: " & myBuf->BPP  
Print "Image Pitch: " & myBuf->Pitch  
Print ""
```

This is what I was talking about earlier. FB will treat your pointer as if it's directly. Since we have the size of the image as well as its pixels address pointer to our screen buffer! See `ScrPtr vs ImgBuf.bas` for an example o

FBGfx Font Header

Header Details

The first row of an image buffer that will be used as a font contains the h (remember that the first row of pixels are going to be the first bytes since

The very first byte tells us what version of the header we're using. Current been released. The second byte tells us the first character supported in

0; Byte; Header Version
1; Byte; First Character Supported
2; Byte; Last Character Supported
3 to (3 + LastChar - FirstChar); Byte; Width of each Character in our font

Creating a Font Buffer

If you had a font that supported character 37 as the first, and character 2

0 for the header version. It's the current only version supported.

37 for the first character supported.

200 for the last character supported.

94 bytes containing the widths of each character.

Since the first row is taken up for header data, the font buffer will be an i if you have a font height of 8, you need a buffer height of 9. You'll be put first as you usually would.

Here's an example (Example3.bas), which creates a font buffer. It only c font:

```
' ' The first supported character
Const FirstChar = 32
' ' Last supported character
Const LastChar = 190
' ' Number of characters total.
Const NumChar = (LastChar - FirstChar) + 1
```

These constants help us. It makes the code cleaner and faster.

```
' ' Create a font buffer large enough to hold 96 ch
' ' Remember to make our buffer one height larger t
Dim As FB.Image Ptr myFont = ImageCreate( ( NumChar
```

Create our font buffer. Remember, we need to add horizontal space for (add an extra row for our font header information.

```
'' Our font header information.  
'' Cast to uByte ptr for safety and consistency, 1  
Dim As UByte Ptr myHeader = Cast(UByte Ptr, myFont )
```

Get the exact, casted, and having no warnings address of our font buffer on this with an fb.Image type.

```
'' Assign font buffer header.  
'' Header version  
myHeader[0] = 0  
'' First supported character  
myHeader[1] = FirstChar  
'' Last supported character  
myHeader[2] = LastChar
```

Assign the header information described above, into the first three bytes last supported character.

```
'' Assign the widths of each character in the font  
For DoVar As Integer = 0 To NumChar - 1  
'' Skip the header, if you recall  
myHeader[3 + DoVar] = 8  
Next
```

Each character in our font can have its own width, so we have to assign starts at 0, so the first time it runs through that code, we'll be at index 3.

```
' ' Remember to destroy our image buffer.  
ImageDestroy( myFont )
```

Just reminding you :D

Assigning Font Characters

This is fairly simple. We'll use FreeBASIC's default font to draw onto our column 0, as the very first column is reserved for header data. Start the and give it the color you want. Be warned, you can't have custom colors buffer, it's stuck the color you draw it as! See the tips & tricks section on

Here's the modified code (Example4.bas), where we'll add the font draw

```
' ' NEW!!!  
' ' Our current font character.  
Dim As UByte CurChar
```

Just to have a quick index of the current ASCII character we're drawing

```
Draw String myFont, ( DoVar * 8, 1 ), Chr(CurChar),
```

Skip the first row of our image buffer, as that contains font buffer informa it with a random color. You should note that we're drawing right into our l

```
Print Chr(CurChar);
```

Just for clarity, so you can see the characters we're drawing into the buffer.

```
'' Use our font buffer to draw some text!  
Draw String (0, 80), "Hello!", , myFont  
Draw String (0, 88), "HOW ARE ya DOIN Today?! YA DOIN  
Sleep
```

Test out our new font. Of course, it's the same one we're used to. You can see it somewhere.

Tips & Tricks

Coloring Your Custom Fonts

Alright, so by now you have realized that once you color a custom font, you can't change the color. We can get around that (CustFontCol.bas). It might be a bit slow, however.

We can create a font object, which has a function to return a font buffer. This function changes the color, and returns the font buffer stored in the object. This *color* function is used to redraw, so we could only redraw from the lowest to the highest. Figure 1 shows the code for this.

```
#include "fbgfx.bi"  
  
Type Font  
    '' Our font buffer.  
    Buf      As FB.Image Ptr  
    '' Font header.  
    Hdr      As UByte Ptr  
  
    '' Current font color.
```

```

Col      As UInteger

    ' Make our font buffer.
Declare Sub Make( ByVal _Col_ As UInteger = RGB(255, 255, 255) )
    ' Change the font color and edit the font buffer.
    ' Return the new font.
Declare Function myFont( ByVal _Col_ As UInteger = _Col_ ) As UInteger

    ' Create/Destroy our font.
    ' Set a default color to it if you like.
Declare Constructor( ByVal _Col_ As UInteger = RGB(255, 255, 255) )
Declare Destructor()
End Type

    ' Create our font's buffer.
Constructor Font( ByVal _Col_ As UInteger = RGB(255, 255, 255) )
    This.Make( _Col_ )
End Constructor

    ' Destroy font buffer.
Destructor Font()
    ImageDestroy( Buf )
End Destructor

    ' Assign the FBgfx font into our font buffer.
Sub Font.Make( ByVal _Col_ As UInteger = RGB(255, 255, 255) )
    ' No image buffer data. Create it.
    If This.Buf = 0 Then

        ' No screen created yet.
        If ScreenPtr = 0 Then Exit Sub

        ' Support 256 characters, 8 in width.
        ' Add the extra row for the font header.
        This.Buf = ImageCreate( 256 * 8, 9 )

        ' Get the address of the font header,
        ' which is the same as getting our pixel address.
        ' Except that we always will use a ubyte.

```

```

This.Hdr = Cast(UByte Ptr, This.Buf) + SizeOf(FE

    '' Assign header information.
This.Hdr[0] = 0
    '' First supported character
This.Hdr[1] = 0
    '' Last supported character
This.Hdr[2] = 255
Else
    If This.Col = _Col_ Then Exit Sub

End If

    '' Draw our font.
For DoVar As Integer = 0 To 255
    '' Set font width information.
    This.Hdr[3 + DoVar] = 8

    Draw String This.Buf, (DoVar * 8, 1), Chr(DoVar)
Next

    '' Remember our font color.
This.Col = _Col_
End Sub

    '' Get the buffer for our font.
    '' Remake the font if the color's different.
Function Font.myFont( ByVal _Col_ As UInteger = RGB(
    '' If our colors match, just return the current
    If _Col_ = Col Then
        Return Buf
    End If

    '' Make the font with a new color.
This.Make( _Col_ )
    '' Return out buffer.
Return This.Buf
End Function

```

```

    ' ' MAIN CODE HERE!
ScreenRes 640, 480, 32

    ' ' Create our font.
Dim As Font myFont = RGB(255, 255, 255)

    ' ' Draw a string using our custom font.
Draw String (0,0), "Hello. I am the custom font.",,
    ' ' Gasp. A new color!
Draw String (0,8), "Hello. I am the custom font.",,
Sleep

    ' ' Speed test. Turns out it's quite slow.
Scope
    Randomize Timer
    ' ' Our timer.
    Dim As Double T = Timer

    ' ' Time how long it takes to make a new font th
For DoVar As Integer = 0 To 499
    myFont.Make( RGB(Rnd * 255, Rnd * 255, Rnd * 255)
Next

    ' ' And we're all done. Print important data.
Locate 3, 1
Print "Time to Re-Draw font 499 times: " & ( Timer
Print "Time per Re-Draw: " & ( Timer - T ) / 500
Sleep
End Scope

```

ScrPtr vs ImgBuf

Comparison of how to draw onto image buffer pixels, versus how to draw

```
#include "fbgfx.bi"
```

```
ScreenRes 640, 480, 32
```

```
'' Create a buffer the size of our screen.  
Dim As FB.IMAGE Ptr myBuf = ImageCreate( 640, 480 )
```

```
'' Get the address of our screen's buffer.  
Dim As uLong Ptr myScrPix = ScreenPtr  
'' Get the address of our pixel's buffer.  
Dim As uLong Ptr myBufPix = Cast( uLong Ptr, Cast( U
```

```
'' Lock our page. Fill the entire page with white  
ScreenLock
```

```
'' Alternatively, if the screen resolution's unknown  
'' make this more secure
```

```
'' Note: this code assumes no padding between rows  
'' you need to use ScreenInfo to get the screen's  
'' row offsets using that instead.
```

```
For xVar As Integer = 0 To 639  
  For yVar As Integer = 0 To 479  
    myScrPix[ ( yVar * 640 ) + xVar ] = RGB(255, 255, 255)  
  Next  
Next
```

```
ScreenUnlock  
Sleep
```

```
'' Draw onto our image buffer all red.  
For xVar As Integer = 0 To myBuf->Width - 1  
  For yVar As Integer = 0 To myBuf->Height - 1  
    myBufPix[ ( yVar * (myBuf->Pitch \ SizeOf(*myBuf  
  Next  
Next
```

```

    '' Put the red buffer on the screen.
Put (0,0), myBuf, PSet
Sleep

/'
    ScreenPtr:
1) Get address of screen buffer
    (remember that FBgfx uses a dummy buffer that it
2) Lock page
3) Draw onto screen address
4) Unlock page to show buffer

    Image Buffer:
1) Create an image buffer
2) Get the address of image pixels
3) Draw onto image pixels
    (you can use neat stuff like the buffer informat
4) Put down Image where you please
    (another big plus!)

    About Drawing:
cast(ubyte ptr, mybuff) + Y * Pitch + X * Bpp

Every Y contains PITCH number of bytes. In order to
have to skip an entire row.

It should be safe to do the pointer arithmetic in c
type is not one byte long, so you may find it easier
match your bit depth.
In these cases you should divide the Pitch and BPP
Conveniently, in this case the Pitch should always
size. And, obviously, so will the BPP, which will ;

'/'

```

Each fbc supports all targets

Since fbc version 0.24, the FreeBASIC compiler always supports all compilation targets. There no longer is any configuration necessary to support for additional targets at fbc compile-time, like it used to exist in c fbc versions. This means you only need to install one fbc per host system and it can be used to compile native programs aswell as non-native programs.

- default: compile for native system
- **-target** and **-arch** compiler options allow cross-compiling

Requirements for cross-compiling

The official FB release packages include an fbc capable of cross-compil but fbc alone is not enough.

1. Besides fbc, FreeBASIC consists of the FB runtime library (rtlib/libfb) , the FB graphics library (gfxlib2/libfbgfx). Additionally, FreeBASIC uses libraries from the MinGW, DJGPP or Linux GCC toolchains. All these lib are precompiled for a certain target. You need a copy of the proper librai for every compilation target you want to use.

2. FreeBASIC uses the assembler and linker (and sometimes even more tools) from the GNU binutils project to create binaries, and these may or support one target at a time. Depending on how they were built, they can support multiple targets. Either way, you need the proper binutils for eve compilation target you want to use.

To keep the official FB release packages small, they only include the libr and tools needed for native development, but not for cross-compiling.

Example: Cross-compiling from Ubuntu GNU/Linux to Win32

Ubuntu offers official MinGW cross-compiling packages, which we can a use for FreeBASIC. The following describes the steps needed to set this

1. gcc/binutils cross-compiler toolchain

Install the `gcc-mingw-w64` package and its dependencies. The exact pack name could be different for different versions of Ubuntu. This should give you the gcc cross-compiler toolchain for targetting Win32 (and Win64 -- you can install the exact packages manually if you prefer to avoid installing the `w64` `gcc-mingw-w64` and all of its dependencies.).

That includes the binutils and MinGW libraries, both of which fbc definitely needs for cross-compiling. It also includes the cross-compiling gcc, which fbc uses to look up the installation locations of the MinGW libraries. Besides gcc is obviously also needed if you want to use `-gen gcc` (such as when targetting 64bit which is currently only supported via `-gen gcc`).

The installed tools are called `i686-w64-mingw32-as` (MinGW cross assembler), `i686-w64-mingw32-ld` (MinGW cross linker), `i686-w64-mingw32-gcc` (MinGW cross gcc), etc. You can use them with fbc by specifying the common target prefix to the fbc **-target** option:

```
fbc foo.bas -target i686-w64-mingw32
```

This tells fbc to cross-compile using the system's `i686-w64-mingw32` gcc/binutils toolchain and libraries.

2. Win32 FB libraries

Install Win32 FB libraries such that fbc can find them. For the `-target i686-w64-mingw32` example from above, the directory where the Win32 FB libraries need to be is `/usr/local/lib/freebasic/win32/`, assuming fbc is installed in `/usr/local/bin/fbc`. You have two options to get them.

a) Copy the libraries from the official Win32 FB release package (or some other existing Win32 build of FB). Create the `/usr/local/lib/freebasic/win32/` directory and copy the libraries into it. This should be safe as long as the Win32 FB libraries are from the same FB

version as the FB-linux setup you have installed. However, if the Win32 libraries were created with a MinGW toolchain that is incompatible with the one from Ubuntu, then there can be errors.

b) Compile the Win32 FB libraries manually using Ubuntu's toolchain. Assuming you have the FB source code in `fb/`, you can do:

```
cd fb
make rtlib gfxlib2 TARGET=i686-w64-mingw32
sudo make install-rtlib install-gfxlib2 TARGET=i686-w64-mingw32
```

This should cross-compile the Win32 FB libraries using the `i686-w64-mingw32` toolchain and install them into the proper directory in `/usr/local`. Again, it is important to ensure that the used source code matches the version of the installed FB-linux setup.

To be completely safe and avoid FB version incompatibilities, you can build an entire FB setup from sources, including the Win32 cross-compiling libraries:

```
cd fb
make
make rtlib gfxlib2 TARGET=i686-w64-mingw32
sudo make install
sudo make install-rtlib install-gfxlib2 TARGET=i686-w64-mingw32
```

Installing gcc for -gen gcc



Windows 32bit

If you are using the FreeBASIC-x.xx.x-win32 package, you can use our pre-made gcc package. Download gcc-x.x.x-for-FB-win32-gengcc.zip from the [Binaries - Windows/More/](#) directory at the fbc downloads area and extract it into the FreeBASIC installation directory (where fbc.exe is) such that gcc.exe and cc1.exe will be placed in these locations:

- bin\win32\gcc.exe
- bin\libexec\gcc\i686-w64-mingw32\x.x.x\cc1.exe

You can also download Win32 versions of gcc directly from the MinGW.org or MinGW-w64 projects.

Windows 64bit

The FreeBASIC-x.xx.x-win64 package already comes with gcc included, and uses -gen gcc by default (because -gen gas does not support 64bit)

DOS

It requires a (minimal) DJGPP installation. DJGPP can be downloaded from the [DJGPP website](#). At least the djdev*.zip and gcc*b.zip are needed. In order to run the DJGPP gcc, the DJGPP environment variable must be set to point to the djgpp.env file.

To use the DJGPP gcc with the FreeBASIC-x.xx.x-dos package, copy gcc.exe and cc1.exe into the FreeBASIC installation directory, such that they will be placed in these locations:

- bin\dos\gcc.exe
- bin\libexec\gcc\djgpp\x.xx\cc1.exe

Linux

Typically the `gcc` package is already installed, or it can be installed by doing something like:

```
sudo apt-get install gcc
```

(the exact command depends on your GNU/Linux distribution)

Non-standalone fbc installed into DJGPP/MinGW toolchains

If you are using a non-standalone version of `fbc` (e.g. from one of the `fbc-x.xx.x-win32` packages), and have it installed inside a DJGPP or MinGW toolchain, then `-gen gcc` should already work, as the DJGPP or MinGW toolchains provide `gcc`.

As long as `gcc.exe` is in the same directory as `fbc.exe` (typically `C:\DJGPP\bin\` or `C:\MinGW\bin\`), or available in the `PATH` environment variable, `fbc.exe` should be able to find and use it.

See also

- `-gen <backend>`

Normal vs. Standalone FreeBASIC



When built from source, FreeBASIC can be configured for and installed one of these two different setups:

Normal build (default)

Standalone build

Normal directory layout:

- bin/
 - fbc.exe
 - [<target>-]ld.exe
 - *other tools for native/cross compilation...*
- include/
 - freebasic/
 - fbgfx.bi
 - *other headers...*
- lib/
 - freebasic/
 - <target>/
 - libfb.a
 - *other libraries...*

Standalone directory layout:

- bin/
 - <target>/
 - ld.exe
 - *other tools...*
- inc/
 - fbgfx.bi
 - *other headers...*
- lib/
 - <target>/
 - libfb.a
 - *other libraries*
- fbc.exe

Differences to the standalone build:

- fbc is located in bin/, like other programs
- looks for includes in include/freebasic/, instead

Differences to the normal build:

- the fbc binary is located at the toplevel, not inside bin/
- looks for tools inside bin/<target>/, i.e. it uses

of `inc/`, to cleanly separate FB headers from system headers

- looks for its own libraries in `lib/freebasic/` instead of `lib/`, to cleanly separate FB libraries from system libraries
- looks for `binutils/gcc` 1) in `bin/` and 2) by relying on `PATH`
- looks for `crt/gcc` libraries 1) in `lib/freebasic/` and 2) by running `"gcc -print-file-name=..."`
- **-target** option accepts system triplets such as `"i686-pc-linux-gnu"` or `"x86_64-w64-mingw32"`
- the target name given to the **-target** option is prepended to the `gcc/binutils` program names when cross-compiling
- compatible with the standard `/usr` or `/usr/local` directories
- typically used for the FB-linux release
- uses `windres` from `binutils` to compile win32 resource scripts

This makes the normal FB build integrate with GNU/Linux distributions and other Unix-like systems pretty well, allows `fbcc` to be

`bin/<target>/ld.exe` instead of `bin/[<target>-]ld.exe`

- looks for FB includes in `include/` not in `include/freebasic/`
- looks for libraries in `lib/`, in `lib/freebasic/`
- does not try to rely on `PATH` and use system tools
- does not try to query `gcc` find files
- **-target** only accepts simple FB target names, no system triplets
- typically used for the FB-linux and FB-win32 releases
- uses `GoRC` to compile win32 resource scripts

The standalone build is intended to be used for self-contained installations such as the traditional FB-win32 and FB-dos releases. It also allows adding `fbcc` to the `PATH` without having to add the whole `bin/` directory.

installed into MinGW or DJGPP trees next to gcc, and allows fbc to work with binutils/gcc cross-compiling toolchains.

Microsoft QuickBASIC



A BASIC compiler, interpreter and IDE

QuickBASIC is a twenty year old interpreter/compiler upon which FreeBASIC is modeled. It runs in 16-bit MS-DOS.

More information from [Wikipedia](#):

Microsoft QuickBASIC (often shortened, correctly, to QB, or incorrectly, to "QBasic", which is a different system) is a descendant of the BASIC programming language that was developed by the Microsoft Corporation for use with the MS-DOS Operating System. It was loosely based on GW-BASIC but in addition provided user-defined types, improved programming structures, better graphics and disk support and a compiler in addition to the interpreter. Microsoft sold QuickBASIC as a commercial development suite.

Microsoft released the first version of QuickBASIC on August 18, 1985 stored on a single 5.25" floppy disk. QuickBASIC came with a markedly different Integrated Design Environment (IDE) from the one supplied with previous versions of BASIC. Line numbers were no longer needed since users could insert and remove lines directly via an onscreen text editor.

Microsoft's "PC BASIC Compiler" was included which could be used to compile programs into DOS executables. The editor also had an interpreter built in which would run the program without leaving the editor at all, and could be used to debug the program before creating an executable file. Unfortunately there were some small, subtle differences between the interpreter and the compiler, so that sometimes programs running perfectly well in the interpreter would fail after compilation, or even not compile at all.

The last version of QuickBASIC was 4.5 (1988) although there was continued development of the Microsoft Basic Professional Development System (PDS), the last release of which was version 7.1 (June 1990). The PDS version of the IDE was called QuickBASIC Extended (QBX). The successor to QuickBASIC and PDS was Visual Basic for MSDOS

1.0 provided in Standard and Professional versions. Later versions of Visual Basic did not include DOS versions as Microsoft wanted developers to concentrate on Windows applications.

A replacement for GW-BASIC, based on QuickBASIC 4.5 was included with MS-DOS 5 and later versions. This is called QBASIC. Compared to QuickBASIC, it is limited as it lacks a few functions, can only handle programs of a limited size, lacks support for separate modules, and is an interpreter only. It cannot be used to produce executable files directly although programs developed using it can still be compiled by a QuickBASIC 4.5, PDS 7.1 or VBDOS 1.0 compiler, if one is available.

To learn more about the language, history, and community of QuickBASIC and its free interpreter-only counterpart, you should see also en.wikipedia.org/wiki/QBasic. There are more links, and more information, including a barebones tutorial for Quick/QBasic programming.

External links

[Pete's QB Site](#). One of the oldest remaining QB sites (since Oct 1998).

[QQN/QBN: QBasic/QuickBasic News](#).

[QQN's Newbies Section](#) which includes a link for downloading QBasic.

Credits (in alphabetical order)



Project Members

- **Andre Victor T. Vicentini** (av1ctor[at]yahoo.com.br):

Founder, main compiler developer, author of many parts of the runtime, FB headers (FBSWIG)

- **Angelo Mottola** (a.mottola[at]libero.it):

Author of the FB graphics library, built-in threads, thread-safe, runtime, ports I/O, dynamic library loading, Linux port.

- **Bryan Stoeberl** (b_stoeberl[at]yahoo.com):

SSE/SSE2 floating point math, AST vectorization.

- **Daniel C. Klauer** (daniel.c.klauer[at]web.de):

FB releases since 0.21, C & LLVM backends, 64bit port, dynamic arrays in UDTs, virtual methods, preprocessor-only mode, miscellaneous fixes and improvements.

- **Daniel R. Verkamp** (i_am_drv[at]yahoo.com):

DOS, XBox, Darwin, *BSD ports, DLL and static library automation, VB-compatible runtime functions, compiler optimizations, miscellaneous fixes and improvements.

- **Ebben Feagan** (sir_mud[at]users.sourceforge.net):

FB headers, C emitter

- **Jeff Marshall** (coder[at]execulink.com):

FB releases since 0.17, FB documentation (wiki maintenance, fbdocs, offline-docs generator), Gosub/Return, profiling support, dialect, specifics, DOS serial driver, miscellaneous fixes and improvements.

- **Mark Junker** (mjscod[at]gmx.de):

Author of huge parts of the runtime (printing support, date/time, function. SCR/LPTx/COM/console/keyboard I/O), Cygwin port, first FB installer scripts.

- **Matthew Fearnley** (matthew.w.fearnley[at]gmail.com):

Print Using & Co, ImageInfo, and others, dialect specifics, optimization improvements in the compiler, many fixes and improvements.

- **Ruben Rodriguez** (rubentbstk[at]gmail.com):

Var keyword, const specifier, placement new, operator overloading and, other OOP-related work, C BFD wrapper, many fixes and improvements

- **Simon Nash:**

AndAlso/OrElse operators, ellipsis for array initializers, miscellaneous fixes and improvements.

Contributors

- **1000101:**

gfxlib2 patches, e.g. image buffer alignment

- **Abdullah Ali** (voodooattack[at]hotmail.com):

Windows NT DDK headers & examples

- **AGS:**

gdbm, zlib, Mini-XML, PCRE headers

- **Claudio Tinivella** (tinycla[at]yahoo.it):

Gtk tutorials

- **Chris Davies** (c.g.davies[at]gmail.com):

OpenAL headers & examples

- **Dinosaur:**

CGUI headers

- **D.J.Peters:**

ARM port, ODE headers & examples, Win32 API header fixes

- **Dumbledore:**

wx-c headers & examples

- **dr0p** (dr0p[at]perfectbg.com):

PostgreSQL headers & examples

- **Edmond Leung** (leung.edmond[at]gmail.com):

SDL headers & examples

- **Eric Lope** (vic_viperph[at]yahoo.com):

OpenGL & GLU headers & examples, examples/gfx/rel-*.bas demos

- **Florent Heyworth** (florent.heyworth[at]swissonline.ch):

Win32 API sql/obdc headers

- **fsw** (fsw.fb[at]comcast.net):

Win32 API headers, Gtk/Glade/wx-c examples

- **Garvan O'Keefe** (sisophon2001[at]yahoo.com):

FB ports of many NeHe OpenGL lessons, PDFlib examples

- **Hans L. Nemeschkal** (Hans.Leo.Nemeschkal[at]univie.ac.at):

DISLIN headers

- **Jofers** (spam[at]betterwebber.com):

ThreadCall keyword, libffi/libjit headers, FreeType examples

- **Jose Manuel Postigo** (postigo[at]uma.es):

Linux serial devices support

- **Laanan Fisher** (laananfisher[at]gmail.com):

FB test suite using CUnit

- **Matthew Riley** (pestery):

OpenGL, GLFW, glexth, FreeGLUT, cryptlib headers

- **Matthias Faust** (matthias_faust[at]web.de):

SDL_ttf headers & examples

- **Marzec:**

SDL headers, SDL_bassgl, SDL_opengl and SDL_key examples, First file routines for FB's rlib

- **MJK:**

big_int header fixes

- **MOD:**

wx-c, BASS headers; -lang qb support for built-in macros, "real" Rnd() algorithm

- **Nek** (dave[at]nodtveidt.net):

Win32 API headers

- **Plasma:**

FMOD and BASS headers & examples

- **Randy Keeling** (randy[at]keeling.com):

GSL matrix example

- **Saga Musix** (Jojo):

BASS examples with sounds

- **Sisophon2001:**

gfxlib2 fixes, Nehe OpenGL lesson ports

- **Sterling Christensen** (sterling[at]engineer.com):

Ex-project-member, author of FB's initial QB-like graphics library

- **TeeEmCee:**

gfxlib2 fixes

- **TJF** (Thomas.Freiherr[at]gmx.net):

ARM port, GTK+, glib, Cairo, Pango headers & examples,

SQLiteExtensions headers

- **zydon:**

Win32 API examples

Greetings

- **Plasma:**

Owner of the freebasic.net domain and main site hoster, many thanks to him.

- **VonGodric:**

Author of the first FreeBASIC IDE: FBIDE.

- **Everybody that helped writing the documentation** (and in special Nexinarus who started it):

<http://www.freebasic.net/wiki/wikka.php?wakka=ContributorList>

- All users that reported bugs, requested features and as such helped improving the compiler, language and run-time libraries.

BLOAD/BSAVE text mode work-around



These functions allow you to use BSAVE and BLOAD in a text mode.

```
Sub _bsave( file As String, p As Any Ptr, sz As Integer )

    Dim As Integer ff
    ff = FreeFile

    Open file For Binary As ff
        fb_fileput( ff, 0, ByVal p, sz )

    Close

End Sub

Sub _bload( file As String, p As Any Ptr )

    Dim As Integer ff
    ff = FreeFile

    Open file For Binary As ff
        fb_fileget( ff, 0, ByVal p, LOF( ff ) )

    Close

End Sub
```

Daniel Verkamp (DrV)

i_am_drv [at] yahoo [dot] com

i_am_drv [at] users [dot] sf [dot] net

<http://drv.nu/>

Part of the FreeBASIC Development team; DOS port maintainer

Creating FB bindings for C libraries



This page aims to document the problems and solutions commonly encountered when creating FB bindings for C libraries.

In general, FB and C/C++ are very similar. FB follows the same ABI as C/C++ as far as possible. The language syntax is also similar to C/C++. As a result, a 1:1 mapping exists between C and FB. However, there are also constructs which cannot be mapped 1:1. FB has function pointer types, but not plain function types.

- The good news: We have tools (**fbfrog** and **h_2_bi**) which can do a lot of the heavy lifting.
- The bad news: There always are some problems which cannot be solved easily.

Data types

C/C++ type	Size in bytes (GCC on Linux/Windows)	Corresponding FB type
char	1	Byte
short [int]	2	Short
int	4	Long
enum (underlying type int)	4	Long
long long [int]	8	LongInt
float	4	Single
double	8	Double
long double	12 on 32bit, 16 on 64bit	CLongDouble
_Bool/bool	1	Byte / Bool
* (pointer)	4 on 32bit, 8 on 64bit	Ptr/Pointer
ssize_t, intptr_t	4 on 32bit, 8 on 64bit	Integer
size_t, uintptr_t	4 on 32bit, 8 on 64bit	UInteger
long [int]	4 on 32bit systems and Win64 (!), 8 on 64bit Linux/BSD	CLong from C

- **Caveat:** int/long is not Integer/Long. In FB, Integer corresponds to C's int (32bit everywhere). Long stays 32bit everywhere. In C, int stays 32bit everywhere, but not on Win64, where long is still 32bit. On Win64, long and C's long are compatible to FB's Integer.
- **Caveat:** long int is not LongInt. FB's LongInt corresponds to C's long long.

- `int` can be translated to `Long`, as both are 32bit consistently.
- `ssize_t` or `intptr_t` can be translated to `Integer` because they type
- `long` cannot be translated directly, but we have `crt/long.bi` which
- `long double` cannot be translated directly, but we have `crt/longd`
- `enum` is a special case. Typically their underlying type is `int` (32bit) changing that. Thus `enums` (used as data type in declarations) can

For example:

```
Enum MyEnum {
    A,
    B
}
```

has to be translated as:

```
Type MyEnum As Long
Enum
    A
    B
End Enum
```

- `BOOL` from `windows.h` is just a typedef for `int`, and should not be converted

Symbol name conflicts

- C/C++ is case-sensitive, with ~50 keywords
- FreeBASIC is case-insensitive, with ~400 keywords
- C code sometimes uses FB keywords as symbol identifiers, for example
- C code often contains identifiers which differ only in case, for example
- In C, a macro can have the same identifier as a function. This is rare

Examples

C code using FB keywords as identifiers:

```
typedef Int Int;  
void Open(void);
```

```
Type INT_ As Long  
Declare Sub open_ cdecl Alias "open"()
```

C code relying on case-sensitivity:

```
void foo(void);  
void Foo(void);  
void FOO(void);
```

```
' ' Wrong translation:  
Extern "C"  
    Declare Sub foo()  
    Declare Sub Foo() ' ' error: duplicate definition  
    Declare Sub FOO() ' ' error: duplicate definition  
End Extern
```

```
' ' Correct translation:  
Extern "C"  
    Declare Sub foo()  
    Declare Sub Foo_ Alias "Foo"()  
    Declare Sub FOO__ Alias "FOO"()  
End Extern
```

Another classic example where this kind of conflict happens:

```
#define GET_VERSION_NUMBER 123  
Int get_version_number(void);
```

```
Extern "C"
    #define GET_VERSION_NUMBER_123 ' renamed to a
    Declare Function get_version_number() As Long
End Extern
```

Conflict between procedure and macro:

```
void f(Int);
#define f(i) f(i + 1)
```

```
Extern "C"
    Declare Sub f(ByVal As Long)
    #define f_(i) f(i + 1) ' renamed to avoid conflict
End Extern
```

Solutions

- Symbols should be renamed by appending _ underscores. API.
- Renaming a symbol should not cause further renames (for foo should be renamed to foo__ instead)
- A list of renamed symbols should be available in the bindir such differences to the original API.
- Fields inside structures do not need to be renamed just because "Name" syntax they can be use FB keywords as identifiers. 7 not a class.

```
Type UdtWithKeywordFields
    As ZString Ptr String ' Field "String" of type
    As Long Type ' Field "Type" of type "Long"
    As Long As ' Field "As" of type "Long"
End Type
```

Function types

In C it's possible to have typedefs with function types. Dereferencing a function pointer types, but not function types.

```
// A Function typedef (Function result = void, no parameters)
typedef void F(void);

// Using it To Declare a Function called f1
F f1;

// Usually f1 would be declared like This (use of Function typedef)
void f1(void);

// A more Common use For Function typedefs Is To Declare a Function Pointer
Extern F *pf1;
```

Since FB does not have function types, such typedefs have to be solved

```
Extern "C"

Type F As Sub() ' Function pointer type

' Declaring procedures is only possible with Declare
Declare Sub f1()

' But at least FB has function pointer types.
' Since F already is the function pointer in the FB
Extern pf1 As F

End Extern
```

Compiling a Development Version of FreeBASIC



The source code of FreeBASIC is maintained on Sourceforge using the Git version control system, which allows different developers to work on the source code at the same time and later combine their work. It is possible for users to download the FreeBASIC source code using anonymous read access and compile it using GNU development tools.

Compiling the development version is not recommended for most users. FreeBASIC is a self-hosting compiler, still in active development, so there will be times when the current development version cannot be compiled by the last official release. Note also that the procedures for building the compiler described here may change with future versions of FreeBASIC.

Essentially, FreeBASIC consists of two parts:

- The FreeBASIC compiler, written in FreeBASIC (self-hosting). Compiling this requires a working FreeBASIC installation.
- The FreeBASIC runtime libraries, written in C. Compiling this requires a C compiler such as gcc, the GNU C compiler (Native gcc on Linux, MinGW on Windows, DJGPP for DOS).

Generally, when compiling FB, care should be taken to never mix compiler and rlib of different versions, because they will not necessarily be compatible. fbc's code generation expects a specific libfb version. Thus, an FB setup should always have the proper libfb version in its `lib` directory, matching the version of the `fbc.exe`. When building a new compiler, just like any other FB program, it will be compiled by an existing fbc and thus it must also be linked against the existing fbc's libfb, not against the new libfb. The new libfb belongs into the new compiler's `lib` directory, not in that of the existing fbc. Typically this means that the compiler should be built first, before rlib/gfxlib2, which is also how the F makefile works by default.

There are two ways to build FB: **normal or standalone**. The normal version is intended for integration with an existing gcc toolchain, while the standalone version makes fbc act more like a self-contained tool. Most importantly, the two use slightly different directory layouts. For example, in the normal version the fbc program is located at `bin/fbc[.exe]`, while

in the standalone version, `fbcc[.exe]` is put into the `toplevel` directory, instead of the `bin/` directory. Furthermore, the directory layout for include files and libraries differs. Traditionally, the FB-linux release is a normal build, while the FB-win32 and FB-dos builds are standalone versions.

Getting the source code

Compiling FB for DOS

Compiling FB on Linux

Compiling FB on Windows

Getting source code updates and recompiling FB

Debugging FB

FB build configuration options

Known problems when compiling FB

GCC toolchain choice

From Git

The FreeBASIC source code is maintained using the **Git version control** system. The source code is available from these Git repositories:

- Main repository at SourceForge:

Git clone URL: `git://git.code.sf.net/p/fbc/code`

Web view: <http://sourceforge.net/p/fbc/code/>

- Mirror repository at GitHub:

Git clone URL: `https://github.com/freebasic/fbc.git`

Web view: <https://github.com/freebasic/fbc>

In order to access a Git repository, you first need to install a Git client.

- Linux:

- The standard Git command line client is available in form of package distributions. For example, on Debian/Ubuntu, you can install it with `apt-get install git git-gui`.
- File explorer integration: Some tools such as **RabbitVCS** provide a graphical command line client. It can integrate into the Nautilus file explorer on Linux and Windows. Install the `rabbitvcs-nautilus` package on Debian/Ubuntu.

- Windows:

- The standard Git command line client is made available for Windows. You can download the latest installer from their website, and install it with `core.autocrlf is true`, so that the FB source code in the windows default MsysGit will add some useful context-menu (right-click) to Windows Explorer.
- There are other Git clients available, for example **Tortoise**

Check out <http://git-scm.com/downloads> for more information.

After installing a Git client, you can download ("clone") the fbc repository

- Using the Git command line in a terminal on Linux:

```
# Clone fbc's SourceForge repository into a new fbc/ directory
git clone git://git.code.sf.net/p/fbc/code fbc

# Open graphical commit history browser:
gitk --all &

# Open graphical commit tool:
git gui &
```

- Using the Git command line in the *Git Bash* terminal that comes v

```
# The Git Bash is an MSYS shell providing a Linux-like command l:
# It should have mapped the ~ home directory to your C:\Document:
# C:\Users\name directory. It is ok to work there, but if you wa
# the fbc repository to somewhere else, you can do so as follows
# Change directory to C:\foo\bar
cd /c/foo/bar

# Clone fbc's SourceForge repository into a new fbc directory
git clone git://git.code.sf.net/p/fbc/code fbc

# Open graphical commit history browser:
gitk --all &

# Open graphical commit tool:
git gui &
```

- Using MsysGit's graphical user interface on Windows: Right click Explorer and select "Git Gui" to bring up the Git Clone window. H fbc repository and the directory into which the clone should go. N directories that already *are* Git repositories will bring up the git-gu
- Other: Please check out your Git client's documentation. No matt probably have to enter the Git clone URL somewhere. Then it sh somewhere on your system.

As a result you should have an fbc/ directory containing the FreeBASIC

repository metadata).

You can regularly update it to the latest version by synchronizing it to the Go into your fbc/ directory and run a Git Pull. When using the Git comman

```
cd fbc/  
git pull
```

From Git but without using a Git client

Both SourceForge and GitHub allow you to download snapshots of the s This way you can download the latest fbc source code without having to generally more efficient though.

- SourceForge: Visit <http://sourceforge.net/p/fbc/code/> in a web bro
- GitHub: Visit <https://github.com/freebasic/fbc> in a web browser ar

Source code for releases

Besides the source code in Git which corresponds to the development v download the source code for the latest official stable release of FreeBA fbc downloads area on SourceForge:

<http://sourceforge.net/projects/fbc/files/>

The source code directory will always contain downloads for the source (source code of previous releases can be found in the older versions dir

The DOS version of FB is typically compiled on a 32bit Windows system with DJGPP and a DOS version of FB installed.

Preparations

Getting the FB source code

To compile a new version of FB, you first need to **get the FB source code**. The following assumes that you have a directory called `fbcdos`, containing the latest FB source code. Naming it `fbcdos` is convenient as avoids conflicts in case you also have an `fb` directory for building the Windows version of FB.

Installing DJGPP

To install DJGPP, we need to download several packages which can be found on the **DJGPP homepage**. FB needs `djdev204.zip` from the **beta/v2/** directory, and several others from the **beta/v2gnu/** directory. If anything is missing from there, you can also look into the **current/v2gnu.** directory. The following packages are needed:

- `binutils (bnu*b.zip)`
- `bash (bsh*.zip)`
- `djdev (djdev*.zip)` - *pick up `djdev204.zip` or later from the `beta/` directory*
- `fileutils (fil*.zip)`
- `gcc (gcc*b.zip)`
- `g++ (gpp*b.zip)`
- `make (mak*b.zip)`
- `shellutils (shl*b.zip)`
- `textutils (txt*b.zip)`

Setup DJGPP by extracting everything into `c:\DJGPP` and adding an environment variable named "DJGPP", set to `c:\DJGPP\djgpp.env`.

It can be useful (especially when working in parallel with MinGW) to use a batch script to launch a terminal with the DJGPP tools in its PATH environment variable, instead of modifying the system's global PATH environment variable:

```
set DJGPP=C:\DJGPP\djgpp.env
set PATH=C:\DJGPP\bin;%PATH%
cd C:\
cmd
```

In the end, you should be able to open a command prompt with `c:\DJGPP\bin` in its PATH, such that running the `gcc` command runs the DJGPP's `gcc` (and not MinGW's `gcc`).

Standalone build (self-contained FB)

Getting an existing FB setup for bootstrapping

We will need a working FB-dos installation to bootstrap the new FB compiler. If you do not have FB-dos installed yet, download the latest FreeBASIC-X.XX.X-dos release from [FB's download site](#). It should be extracted somewhere like `c:\FreeBASIC-X.XX.X-dos`.

Building the new FB setup

If you want to create a **traditional standalone** FB-dos setup like the one from the `FreeBASIC-X.XX.X-dos` release package, you need to tell FB's makefile by setting the `ENABLE_STANDALONE` variable. Assuming the FB sources are located at `c:\fbcdos`, create a `c:\fbcdos\config.mk` file containing the following:

```
ENABLE_STANDALONE = 1
```

Then, open a command prompt with `c:\DJGPP\bin` in its PATH, go to the

directory with the FB source code, run "make" with the `FBC=...` variable set to point to the existing `fbc.exe` to use for bootstrapping, and let it compile:

```
> cd C:\fbcdos
> make FBC=C:/FreeBASIC-X.XX.X-dos/fbc.exe
```

This should have produced the `fbc.exe` compiler and the libraries in `lib\dos\`. To complete this new FB setup, you need to add the binutils (`as.exe`, `ar.exe`, `ld.exe`) into `bin\dos\` and copy in some DJGPP libraries into `lib\dos\`.

- Copy these files to `c:\fbcdos\bin\dos`:
 - `C:\DJGPP\bin\{ar,as,ld}.exe`
- Copy these files to `c:\fbcdos\lib\dos`:
 - `C:\DJGPP\lib\{crt0,gcrt0}.o`
 - `C:\DJGPP\lib\lib{emu,m}.a`
 - `C:\DJGPP\lib\gcc\djgpp\[version]\libgcc.a`

You can copy more libraries if you need them, for example the `C:\DJGPP\lib\gcc\djgpp\[version]\libsupcxx.a` C++ support library, or others from the `C:\DJGPP\lib\` directory.

A note on `libc.a`: FB needs a modified version of DJGPP's `libc.a` because DJGPP's `libc.a` contains a bug (see `contrib/djgpp/readme.txt` from the `fbc` source code for more information). The FB makefile should have taken care of this and produced the modified version of `libc.a` at `lib\dos\libc.a`. This should not be overwritten with DJGPP's original `libc.a`.

Now, the new FB setup should be ready for use. You can use it right from the source tree or copy it somewhere else. The following are the relevant files and directories:

- `fbc.exe`

- bin/dos/
- inc/
- lib/dos/

If you rebuild it in the future (e.g. after updates to the FB source code from Git), you can let it rebuild itself by just running "make" without specifying an external FBC. It will then use the default, FBC=fbc, which in this case corresponds to the fbc.exe in the same directory.

```
> cd C:\fbcdos
> make
```

Normal build (like Linux)

Getting an existing FB setup for bootstrapping

We will need a working fbc installation to bootstrap the new FB compiler. If you do not have fbc installed yet, download the latest fbcXXXXb package from [FB's download site](#), and extract it into the DJGPP directory (C:\DJGPP) like a DJGPP package. This will add a working fbc to your DJGPP installation.

Building the new FB setup

In order to create a normal (non-standalone) build like the one from the fbcXXXXb release package, just compile FB without specifying ENABLE_STANDALONE. Open a command prompt with C:\DJGPP\bin in its PATH, go to the directory with the FB source code, run "make" and let it compile.

```
> cd C:\fbcdos
> make
```

This should have produced the bin/fbc.exe compiler and the libraries in

lib\freebas\dos\.

Optionally, you can copy this setup into the c:\DJGPP tree by running "make install":

```
> make install prefix=C:/DJGPP
```

It can be useful to store the prefix variable in `config.mk`, so you can run `make install` in the future without having to set it manually again:

```
# config.mk:  
prefix = C:/DJGPP
```

Installing `fbcc` into the DJGPP tree this way means that it acts as if it was part of DJGPP. However, it is also possible to use `fbcc` from the source tree, without installing it elsewhere. It will invoke `gcc -print-file-name=...` in order to locate the DJGPP binutils and libraries.

Building FB on Linux is fairly easy because usually the GNU/Linux distributions provide all the needed development packages and they can be installed easily, at least for native builds. Since 64bit support was added to FB, a native build should always be possible, no matter whether you have a 32bit x86 or 64bit x86_64 system. Cross-compiling the 32bit x86 version of FB on a 64bit x86_64 system (or vice-versa) and building for other architectures such as ARM is also possible.

Generally, compiling FB-linux requires the following packages:

- an existing, working FreeBASIC setup for bootstrapping the new compiler
- gcc
- make
- ncurses development headers & libraries (actually only its libtinfo part)
- gpm development headers & libraries (general purpose mouse)
- X11 development headers & libraries (including X11, Xext, Xpm, Xrandr, Xrender)
- OpenGL development headers & libraries (typically from the Mes project)
- libffi development headers & libraries

Native build

Getting the FB source code

To compile a new version of FB, you first need to **get the FB source code**. The following assumes that you have a directory called `fb_c`, containing the latest FB source code.

Getting an existing FB-linux setup for bootstrapping

We will need a working FB-linux installation to bootstrap the new FB

compiler. If you do not have a native version of FB installed yet, download the latest FreeBASIC-X.XX.X-linux release for your system (32bit x86, 64bit x86_64, ARM, etc.) from [FB's download site](#), then extract and install it:

```
$ tar xf FreeBASIC-X.XX.X-linux.tar.gz
$ cd FreeBASIC-X.XX.X-linux
$ sudo ./install.sh -i
```

It is possible that you can get working FB setups from other sources besides the fbc project. For example, some distros may provide freebasic packages out-of-the-box.

Installing development packages

The following lists show the packages you have to install for some common GNU/Linux distributions. The exact package names can be different depending on which distro (or which version of it) you use.

Debian-based systems (including Ubuntu, Mint etc.):

- gcc
- make
- libncurses5-dev
- libgpm-dev
- libx11-dev
- libxext-dev
- libxpm-dev
- libxrandr-dev
- libxrender-dev
- libgl1-mesa-dev
- libffi-dev

OpenSUSE:

- gcc

- make
- ncurses-devel
- gpm-devel
- libX11-devel
- libXext-devel
- libXpm-devel
- libXrandr-devel
- libXrender-devel
- Mesa-libGL-devel
- libffi48-devel

Fedora:

- gcc
- make
- ncurses-devel
- gpm-devel
- libX11-devel
- libXext-devel
- libXpm-devel
- libXrandr-devel
- libXrender-devel
- mesa-libGL-devel
- libffi-devel

Compiling FB

Compiling FB natively is as simple as running "make" in the fbc source code directory. This will build a native FB setup matching the system architecture, assuming that the existing fbc installed on the system produces native programs.

```
$ cd fbc
$ make
```

This should have produced the `bin/fbc` compiler and the libraries in `lib\freebasic\linux-[architecture]\`.

Afterwards, you can install the new `fbc` build into `/usr/local` by running "make install", and overwrite the old FB installation:

```
$ sudo make install
```

Compiling 32bit FB on a 64bit system with existing 32bit FB

Besides native builds, you can also make non-native builds, such as compiling the 32bit version of FB on a 64bit system, using an existing 32bit FB build to bootstrap. This was very common before 64bit support was added to FB. It requires a slightly different procedure than a native build.

- Get the FB source code.
- Install a 32bit version of FB for bootstrapping (instead of a native 64bit version).
- Install 32bit development packages (not just the native 64bit ones).

64bit Debian/Ubuntu example:

- `gcc-multilib`
- `make`
- `lib32ncurses5-dev`
- `libx11-dev:i386`
- `libxext-dev:i386`
- `libxpm-dev:i386`
- `libxrandr-dev:i386`
- `libxrender-dev:i386`
- `libgl1-mesa-dev`
- `libgpm-dev`

- lib32ffi-dev
- 64bit OpenSUSE example:
- gcc-32bit
 - make
 - ncurses-devel-32bit
 - gpm-devel
 - libX11-devel-32bit
 - libXext-devel-32bit
 - libXpm-devel-32bit
 - libXrandr-devel-32bit
 - libXrender-devel-32bit
 - Mesa-libGL-devel-32bit
 - libffi48-devel-32bit
- Add the following `config.mk` file to the `fb` source tree (next to the `FB` makefile):

```
CC = gcc -m32
TARGET_ARCH = x86
```

This tells the `FB` makefile to build for 32bit instead of the 64bit default.

Setting `CC` to `gcc -m32` instead of `gcc` causes all C code to be compiled for 32bit rather than the default 64bit.

Assuming that the existing installed `fb` is a 32bit one, it will already default to compiling to 32bit, so setting `FBC` to `fb -arch 32` instead of `fb` is not needed (and older 32bit-only `fb` versions did not even have the `-arch 32` option anyways).

Setting the `TARGET_ARCH` to `x86` is necessary to override the `FB` makefile's `uname -m` check (because that returns `x86_64` on 64bit). This allows the `FB` makefile to select the proper `x86` `rtlib/gfxlib2` modules and to use the correct directory layout for `x86`.

- Run "make" and let it compile `FB`:

```
$ cd ~/fbc  
$ make
```

- Optionally, install the newly built 32bit FB setup into /usr/local:

```
$ sudo make install
```

Preparations

Getting the FB source code

To compile a new version of FB, you first need to **get the FB source code**. This section assumes that you have a directory called `fb`, containing the latest FB source code.

Installing a MinGW-w64 toolchain

In this guide we will use a 32bit or 64bit **MinGW-w64** toolchain to build the 32bit or 64bit version of FB, respectively. Visit <http://sourceforge.net/projects/mingw-w64> and go to the **Toolchains targetting Win64** or **Toolchains targetting Win32** directory, whether you want to compile a 32bit or 64bit version of FB. Enter the `Per builds/` subdirectory, choose the latest gcc version, then enter the `threads` subdirectory and download the toolchain package from there.

Extract the toolchain into a new `c:\mingw-w64` directory, such that you end up with `c:\mingw-w64\bin\gcc.exe`.

If you know what you are doing, you can also use a different MinGW-w64 one from different projects such as MinGW.org or TDM-GCC. We have some notes on the MinGW toolchain choices on the [DevGccToolchainChoice](#) page.

Installing MSYS

MSYS (originally a Cygwin fork) brings a Unix-like shell environment to Windows. It includes GNU make, the bash shell and Unix command line tools such as `cp` and `rm` to run the FB makefile and the FB test suite.

The needed MSYS packages can be downloaded and extracted by using the **mingw-get setup from the MinGW.org project**.

Run the installer and choose `c:\mingw` as installation directory. This way you avoid conflicts with `c:\mingw-w64`, avoiding potential conflicts. The MinGW Installation Manager (`C:\mingw\bin\mingw-get.exe`) should be opened automatically afterwarward. Then click on the `mingw-developer-toolkit` package from the Basic Setup section by click

package name and selecting "Mark for Installation", then selecting Install Changes from the application's menu.

This should install the commonly needed MSYS components. We do not provide mingw32-base or mingw32-gcc-* packages here, because we are using our own toolchain instead of the MinGW.org one. If you do not wish to use the MinGW/MSYS packages, you can also download the MinGW/MSYS packages manually from the [MinGW/MSYS download site](#).

Ultimately, MSYS should be installed at `C:\MinGW\msys\1.0\`. Now there are three sub-directories: `C:\MinGW-w64\bin\`, `C:\MinGW\bin\` and `C:\MinGW\msys\1.0\bin\`. All three should be added to the PATH environment variable (in the given order), so that the executables will be found when invoked from a command prompt or from the FB makefile.

In order to avoid modifying the system-wide PATH, you can use a batch file named `open-msys.bat` following to open an MSYS bash with the needed PATH settings, every time you need it.

```
set PATH=C:\MinGW\msys\1.0\bin;%PATH%
set PATH=C:\MinGW\bin;%PATH%
set PATH=C:\MinGW-w64\bin;%PATH%
C:\MinGW\msys\1.0\msys.bat
```

Getting libffi

The FB rlib source code depends on **libffi** headers (`ffi.h` and `ffitarget.h`). The gcc toolchains include directory (`C:\MinGW-w64\i686-w64-mingw32\include` for 32bit MinGW-w64 and `C:\MinGW-w64\x86_64-w64-mingw32\include` for 64bit MinGW-w64). A `libffi.a` library will be needed later when compiling FB programs that use libffi.

Prebuilt versions of libffi are available from the [fbc downloads area](#).

If you do not want to use a prebuilt version, but prefer to compile libffi manually, it is fairly simple. libffi uses the autotools (autoconf, automake, libtool) build system. The corresponding packages have to be installed for MinGW/MSYS. Open the command prompt with the proper PATH settings).

- 32bit:

```
$ ./configure
$ make
```

- 64bit: This requires working around MSYS' `uname` which still returns

```
$ ./configure --build=x86_64-w64-mingw32 --host=x86_64-w64-mingw32
$ make
```

This should produce the libffi headers in an `include/` subdirectory and the `.libs/` subdirectory. You can then copy them into the corresponding directory of the w64 toolchain such that `gcc` will find them.

Standalone build (self-contained FB)

Getting an existing FB setup for bootstrapping

We will need a working FB-win32 installation to bootstrap the new FB compiler. If you do not have FB-win32 installed yet, download the latest `FreeBASIC-X.XX.X-win32` from the [download site](#). It should be extracted somewhere like `C:\FreeBASIC-X.XX.X`.

Building the new FB setup

If you want to create a **traditional standalone** FB-win32 setup like the `FreeBASIC-X.XX.X-win32` release package, you need to tell FB's makefile by setting the `ENABLE_STANDALONE` variable. Furthermore, in order to compile for 64bit, you need to set the `TARGET_ARCH` variable manually, because MSYS' `uname -m` command returns `x86_64` and thus the FB makefile would mis-detect the system as 32bit. As the files are located at `C:\fbc`, create a `C:\fbc\config.mk` file containing the following:

- 32bit:

```
ENABLE_STANDALONE = 1
```

- 64bit:

```
ENABLE_STANDALONE = 1
```

```
# Manually set TARGET_ARCH to override uname check for 64bit
TARGET_ARCH = x86_64
```

Then, open the MSYS bash using the .bat script mentioned above (with settings), go to the directory with the FB source code, run "make" with the path to point to the existing fbc.exe to use for bootstrapping, and let it compile

```
$ cd /c/fbc
$ make FBC=C:/FreeBASIC-X.XX.X-win32/fbc.exe
```

This should have produced the fbc.exe compiler and the libraries in lib\win32\ and lib\win64\ respectively. To complete this new FB setup, you need to add the binutils (ld.exe, dlltool.exe) into bin\win32\ and copy in some MinGW libraries into

- Copy to C:\fbc\bin\win32 (32bit) or C:\fbc\bin\win64 (64bit):
 - C:\MinGW-w64\bin\{ar,as,ld,dlltool}.exe
 - GoRC.exe from <http://www.godevtool.com/>
- For 64bit, or for using -gen gcc on 32bit, gcc.exe and cc1.exe are needed:
 - Copy C:\MinGW-w64\bin\gcc.exe to C:\fbc\bin\win{32|64}
 - Copy C:\MinGW-w64\libexec\gcc\[target]\[version]\cc1.exe to C:\fbc\bin\libexec\gcc\[target]\[version]\cc1.exe
- Copy to C:\fbc\lib\win32 (32bit) or C:\fbc\lib\win64 (64bit):
 - C:\MinGW-w64\[target]\lib\{crt2,dllcrt2,gcrt2}.o
 - C:\MinGW-w64\[target]\lib\lib{gmon,mingw32,mingwex,mingwex64}.a
 - C:\MinGW-w64\[target]\lib\lib{advapi32,gdi32,kernel32,msvcrt,user32}.a
(rename to lib*.dll.a if wanted)
 - C:\MinGW-w64\lib\gcc\[target]\[version]\{crtbegin,crtend}.a
 - C:\MinGW-w64\lib\gcc\[target]\[version]\libgcc.a
 - libffi.a (from the prebuilt libffi package or your own build)

([target] refers to i686-w64-mingw32 for 32bit MinGW-w64 or x86_64-w64-mingw32 for 64bit MinGW-w64, and [version] is the gcc version number)

You can copy more libraries if you need them, for example the `c:\MinGW-[version]\libsupc++.a` C++ support library, or other Win32 API DLL into `C:\MinGW-w64\[target]\lib\` directory.

Now, the new FB setup should be ready for use. You can use it right from where it is, or copy it somewhere else. The following are the relevant files and directories:

- `fb.exe`
- `bin/win32/` (32bit) or `bin/win64/` (64bit)
- `inc/`
- `lib/win32/` (32bit) or `lib/win64/` (64bit)

Normal build (like Linux)

Getting an existing FB setup for bootstrapping

We will need a working `fb` installation to bootstrap the new FB compiler. If not installed yet, download the latest `fb-X.XX.X-mingw-w64-i686` (32bit) or `w64-x86_64` (64bit) package from [FB's download site](#), and extract it into a directory (e.g. `C:\MinGW-w64`) like a MinGW package. This will add a working installation.

Building the new FB setup

In order to create a normal (non-standalone) build, just compile FB with `ENABLE_STANDALONE`. However, in order to compile for 64bit it is necessary to set the `uname -m` command manually, because MSYS' `uname -m` command does not support `64bit`. The `makefile` would mis-detect the system as 32bit.

- 32bit: no `config.mk` needed.
- 64bit: Create a `config.mk` containing the following:

```
# Manually set TARGET_ARCH to override uname check for 64bit
TARGET_ARCH = x86_64
```

Then, open the MSYS bash using the `.bat` script mentioned above (with

settings), go to the directory with the FB source code, run "make" and le

```
$ cd /c/fbc
$ make
```

This should have produced the `bin/fbc.exe` compiler and the libraries in `lib\freebasic\win64\` respectively.

Optionally, you can copy this setup into the `c:\MinGW-w64` tree by running

```
$ make install prefix=C:/MinGW-w64
```

It can be useful to store the prefix variable in `config.mk`, so you can run it in the future without having to worry about it:

```
# config.mk:
prefix = C:/MinGW-w64
```

Installing `fbc` into the MinGW tree this way means that it acts as if it was installed. However, it is also possible to use `fbc` from the source tree, without installing it. To do this, you can invoke `gcc -print-file-name=...` in order to locate the MinGW binutils and

Getting source code updates and recompiling FB



To download updates made available in the fbc Git repository, you can do this using a graphical Git tool, or in a terminal:

```
git pull
```

To take a look at incoming changes *before* applying them, do this:

```
# Update remote branches
git fetch

# Take a look
gitk --all

# Everything looks ok? Then merge the remote branch into the current
git merge origin/master
```

Rebuilding is, most of the time, as easy as running "make" again. Of course, if you have any options (like `ENABLE_STANDALONE`) for the build, you have to specify them again in `config.mk`.

```
make
# or if needed:
make ENABLE_STANDALONE=1
```

As a special exception, for the DOS build it is necessary to run `make clean` because some source modules have been renamed or deleted. The reason for this is that the DOS linker does not support `*.o` wildcards to link `fbc` and archive `libfb.a` etc., instead of passing the explicit object files. This is to obey the command line length limitation. If `make clean` is not run, it may use the previous build. Luckily, we do not rename or delete source files often.

Debugging FB



For debugging and development it's a good idea to build the compiler with `-g` and `-exx` to enable assertions and NULL pointer/array boundary checks. For the `rtlib/gfxlib2` code, `-DDEBUG` enables the assertions. Just update `config.mk` and (re)build. Example `config.mk` settings:

```
FBFLAGS := -g -exx
CFLAGS := -g -O0 -DDEBUG
```

Running `fb` inside `gdb` typically looks like this:

```
gdb --args fbc foo.bas
```

Running `fb` inside `valgrind` typically looks like this:

```
valgrind fbc foo.bas
```

Also note that `fb` can be tested right from inside the build tree, without having to be "installed" somewhere else, which also is a great debugging and development help.

FB build configuration options



The FB makefile as well as the compiler/rtlib/gfxlib2 source code offers some configuration options. If you build FB by using the FB makefile, then it makes sense to use the FB makefile's configuration options. If you build FB by compiling the sources manually (without using the FB makefile), then of course you can only use the source code configuration options, and you are responsible for putting the FB setup together properly yourself.

The compiler and rtlib/gfxlib2 source code both handle some #defines which allow for some configuration. For example, #defining `ENABLE_STANDALONE` when building the compiler (by specifying `-d ENABLE_STANDALONE` on the `fbcc` command line) will adjust the compiler for **standalone setup**. As another example, #defining `DISABLE_FFI` when building the rtlib (by specifying `-DDISABLE_FFI` on the `gcc` command line) will cause the rtlib to be built without using the libffi headers (`ffi.h`). This disables **Threadcall** support in the rtlib, but can be useful if you do not have libffi.

When using the FB makefile, you can set some variables on the make command line or inside `config.mk` that affect how the makefile will invoke the `fbcc/gcc` compilers and what directory layout it will use for the FB setup. This includes cases where the makefile will automatically pass the configuration options on to the compiler/rtlib/gfxlib2 source code. For example, specifying `ENABLE_STANDALONE=1` to the FB makefile causes it to use `-d ENABLE_STANDALONE` when building the new compiler (make it standalone) and to put the newly built compiler and libraries into the standalone directory layout.

FB makefile commands

- none or all

The default - builds everything that needs to be built

- compiler, rtlib, gfxlib2

Used to build a specific component only. For example, this can be used to build an `rtlib` for a specific target, in order to be able to cross-compile FB programs (such as the `compiler`) for that target.

- `clean[-component]`

Used to remove built files. `make clean` removes all built files, while for example `make clean-compiler` removes only the files built for the compiler, allowing the compiler to be recompiled more quickly, without the need to rebuild the whole `rtlib/gfxlib2` code.

- `install[-component]`, `uninstall[-component]`

Used to copy the built files into the directory specified by the `prefix` variable, or remove them from there. This is most useful to install the normal build into `/usr/local` on Linux/BSD systems. For the standalone build, `make install` will also work and copy over or remove the files. However, the standalone build uses an incompatible directory layout and should not be installed into `/usr/local` or similar directories because of this.

Note that it is fine to run the newly built FB setup right from the directory where it was compiled; `make install` is not necessary to make it work (unless the `prefix` path was hard-coded into the compiler via `ENABLE_PREFIX`).

Additionally there are `install-includes` and `uninstall-includes` commands, which copy/remove just the FB includes (header files). Note that there is no `make includes` or similar command, as the includes do not need to be built.

FB makefile configuration

The following variables are intended to be set on the `make` command line or inside a file called `config.mk` next to the FB makefile which is read in by the FB makefile. `config.mk` is useful for setting variables in a permanent way such that you do not have to specify them manually everytime when invoking `make`.

Make command line example:

```
$ make CFLAGS=' -O2 -g '
```

config.mk example:

```
CFLAGS = -O2 -g
```

- FBFLAGS, FBCFLAGS, FBLFLAGS

Extra fbc flags to be used when compiling and/or linking the compiler. The default is `-maxerr 1` (check the FB makefile for more details). Typically this is used to add options such as `-g -exx` to build a debug version of the compiler.

- CFLAGS

Extra gcc flags to be used when compiling `rtlib` and `gfxlib2`. The default is `-O2` (check the FB makefile for more details). Typically this is overridden for debugging purposes by doing `CFLAGS=-g`.

- prefix

The FB installation path. The default is `/usr/local`. Note: MSYS maps `/usr/local` to `C:\msys\1.0\local`.

This is only used...

- by the makefile's `install` and `uninstall` commands
- in the compiler (hard-coded) if `ENABLE_PREFIX` was used

Note that in combination with `bash` on Win32 (e.g. from DJGPP or MSYS) it's necessary to use forward slashes instead of backslashes in directory paths, for example: `prefix=C:/MinGW`

- TARGET

This variable can be set to a gcc toolchain triplet such as `i686-pc-linux-gnu` or `x86_64-w64-mingw32` in order to cross-compile using that GCC cross-compiler toolchain. The makefile will use `fbc -target $(TARGET)` instead of `fbc`, and `$(TARGET)-gcc` instead of `gcc`.

For example, on a Debian GNU/Linux system with the `i686-w64-mingw32` GCC cross-compiler installed, you can build the win32 `rtlib` like this:

```
# Build the win32 rtlib/gfxlib2
```

```
make rtlib gfxlib2 TARGET=i686-w64-mingw32

# Install it into /usr/local/lib/i686-w64-mingw32-freebasic
make install-rtlib install-gfxlib2 TARGET=i686-w64-mingw32
```

It will supplement the existing fbc installation in /usr/local, like a plugin, and from now on you can cross-compile FB programs for win32 by doing

```
fbc -target i686-w64-mingw32 ...
```

- FBC, CC, AR

These variables specify the fbc, gcc and ar programs used during the build. You can specify them to override the defaults, for example:

- `make FBC=~/.FreeBASIC-0.90.1-linux/fbc CC="gcc -m32"`

FBC affects the compiler source code only, while cc and ar are used for rtlib and gfxlib2.

- V=1

v for verbose. By default, the makefile does not display the full command lines used during compilation, but just prints out the latest tool and file name combination to give a better visual indication of the build progress. It also makes warnings and errors stand out more in the console window. If the variable v is set, the echoing tricks are disabled and full command lines will be shown, as GNU make normally does.

- ENABLE_STANDALONE=1

Build a standalone FB setup instead of the normal Unix-style setup, see also: [the standalone vs. normal comparison](#). This causes the makefile to use the standalone directory layout and to use `-d ENABLE_STANDALONE` when building the compiler.

- ENABLE_PREFIX=1

This causes the makefile to use `-d ENABLE_PREFIX=$(prefix)` when building the compiler.

- ENABLE_SUFFIX=foo

This causes the makefile to use `-d ENABLE_SUFFIX=$(ENABLE_SUFFIX)` when building the compiler, and to append the given suffix string to the fbc executable's and lib/ directories' names.

For example, using `ENABLE_PREFIX=-0.24` will give you `bin/fbc-0.24.exe` and a `lib/freebasic-0.24/` directory, instead of the default `bin/fbc.exe` and `lib/freebasic/`. This allows installing multiple versions of compiler and runtime in parallel.

Note: The `include/freebasic/` directory name is not affected, and the FB headers are always shared by all installed FB versions (FB's headers and their directory layouts are designed to be able to do that).

This is only supported for the normal (non-standalone) build. It is not needed for the standalone build, because everyone of those can be in a separate installation directory anyways, while normal (non-standalone) builds may have to share a common installation directory such as `/usr/local` or `C:\MinGW`.

- `ENABLE_LIB64=1`

This causes the makefile to use `-d ENABLE_LIB64` when building the compiler. 64bit libraries are placed into `lib64/freebasic/` instead of `lib/freebasic/`.

Compiler source code configuration (FBFLAGS)

- `-d ENABLE_STANDALONE`

This makes the compiler behave as a standalone tool that cannot rely on the system to have certain programs or libraries. See [the normal vs. standalone comparison](#) for more information.

- `-d ENABLE_SUFFIX=foo`

This makes the compiler append the given suffix to the `lib/freebasic/` directory name when searching for its own `lib/freebasic/` directory. For example, `-d ENABLE_SUFFIX=-0.24` causes it to look for `lib/freebasic-0.24/` instead of `lib/freebasic/`. Corresponding the `ENABLE_SUFFIX=foo` makefile option, this adjust the compiler to work in the new directory layout.

- `-d ENABLE_PREFIX=/some/path`

This causes the given prefix path to be hard-coded into the compiler, disabling the use of `Exepath()`. Thus it will no longer be relocatable. This is useful if its known that the compiler does not need to be relocatable, or if `exepath()` does not work properly (for example, in FB 0.90.1, this is the

case for FreeBSD).

- -d ENABLE_LIB64

This makes the compiler search 64bit libraries in `lib64/freebasic/` instead of `lib/freebasic/`. This only affects the normal (non-standalone) build. 32bit libraries are still searched in `lib/freebasic/`.

rtlib and gfxlib2 source code configuration (CFLAGS)

- -DDISABLE_X11

With this, the Unix `rtlib/gfxlib2` will not use X11 headers, disabling `gfxlib2`'s X11 graphics driver and some of the `rtlib`'s Linux console functionality (affects `multikey()` and console mouse handling).

- -DDISABLE_GPM

With this, the Linux `rtlib` will not use General Purpose Mouse headers (`gpm.h`), disabling the Linux **GetMouse** functionality.

- -DDISABLE_FFI

With this, the `rtlib` will not use `libffi` headers (`ffi.h`), disabling the **Threadcall** functionality.

- -DDISABLE_OPENGL

With this, the `gfxlib2` will not use OpenGL headers, disabling the OpenG graphics drivers.

Known problems when compiling FB



Win32 rtlb compilation error: wchar.h: unknown type name 'dev_t'

<http://sourceforge.net/p/mingw/bugs/2039/>

The `wchar.h` header file from MinGW.org contains a `struct _stat64` decl because it uses `dev_t`, `ino_t`, `mode_t` which are only available with an `_FB` rtlb we `#define _NO_OLDNAMES` when compiling.

To work around this issue, adjust `wchar.h` and add `_` underscore prefixes

Win32 rtlb compilation error: _controlfp, _PC_64 undeclared

```
CC src/rtlib/obj/hinit.o
src/rtlib/win32/hinit.c: In function 'fb_hInit':
src/rtlib/win32/hinit.c:21:5: warning: implicit declaration of f
src/rtlib/win32/hinit.c:21:17: error: '_PC_64' undeclared (first
```

Both the MinGW.org runtime and GCC have a `float.h` header, and in so the above errors.

Easiest temporary fix: Append `#include_next <float.h>` to gcc's `float.h`

See also:

- The comments at the top of `c:\MinGW\include\float.h`
- <http://sourceforge.net/p/mingw/bugs/1580/>
- <http://sourceforge.net/p/mingw/bugs/1809/>
- <http://gcc.gnu.org/ml/gcc-patches/2010-01/msg01034.html>

MinGW binutils ld versions 2.18 to 2.21

`fb` triggers a bug ([binutils ld bug 12614](#)) in the mentioned linker version in binutils 2.21.1 and up.

MinGW.org runtime's globbing code changes case of command line

<http://sourceforge.net/p/mingw/bugs/2062/>

MinGW.org's runtime (mingwrt-4.0.3) changed the case of command line existing file/directory name and only differed in case, it was adjusted to r whose command line parsing is not case-insensitive. For example, `gui k fbc.exe's -s gui` option, making it impossible to use, as `fbc.exe` refused t

-lXpm not found on Debian x86 64

The `ia32-libs-dev` package (for example on Debian 6) for some reason c does contain those for the other X11 development libraries. This appare

```
ln -s /usr/lib32/libXpm.so.4 /usr/lib32/libXpm.so
```

DJGPP: Too many open files

If a DJGPP program fails with a *too many open files* error on Windows, t

- Use `msconfig` to add `PerVMFiles=255` to the `[386Enh]` sectic
- Edit the `files=` setting in `C:\WINDOWS\system32\CONFIG.NT:`
- Also see http://www.delorie.com/djgpp/v2faq/faq9_7.html

GCC toolchain choice



FB is based on GCC toolchains and corresponding libraries. However, there is not a single GCC toolchain per platform, but often multiple slightly different ones. FB can generally work with all of them, but still there can be differences depending on the toolchain chosen to build and use FB. Here we document some of the issues to consider when building FB and/or making FB release-ready.

Windows (MinGW)

MinGW toolchains:

- **MinGW.org** - also provides MSYS, besides a MinGW GCC toolchain. It has Win64 support (yet).
- **MinGW-w64** - 32bit and 64bit. Different runtime libraries than MinGW.org.
- **TDM-GCC** - 32bit based on MinGW.org, 64bit based on MinGW-w64 with some modifications.
- MinGW cross-compilers on various GNU/Linux distributions - for example, MinGW-w64 on Debian/Ubuntu and Fedora (i686-w64-mingw32, x86_64-w64-mingw32)

Notes:

- GCC exception handling mechanism: SJLJ setjump/longjump (slow but safe), DWARF-2 (fast but does not always work). The MinGW.org toolchain uses DWARF2, while for MinGW-w64, both types are available. FB does not support exceptions anyways, so in theory the exception handling mechanism used by the underlying GCC toolchain does not matter.

In practice though, DWARF-2 GCC generates static data for stack unwinding which is put into `.eh_frame` sections. The problem is that `.eh_frame` data is also generated for C code (not just C++ code) like all the FB/GCC/MinGW runtime libraries, and it increases `.exe` size noticeably. This can be avoided in multiple ways:

- Use gcc flags to disable the generation of the `.eh_frame` data. FB is using this in its makefile and for `-gen gcc`

however obviously it does not affect the prebuilt MinGW/GCC libraries (unless the entire toolchain is

```
-fno-exceptions -fno-unwind-tables -fno-asynchronous-unwind-tables
```

- Discard/strip the `.eh_frame` section when linking (by custom `ldscript`)
- Use an SJLJ toolchain (i.e. MinGW-w64 built for SJLJ instead of MinGW.org)

Furthermore, the exception handling method may be an important detail (even if you do not care about `.exe` size) if you want to use C++ libraries in case the C++ library uses exceptions.

- GCC threading model: Win32 threads (native), POSIX threads (by `winthreads` library). The MinGW.org toolchain uses Win32 threads for MinGW-w64, both types are available.

GCC needs POSIX threads to implement certain new C++ features, which is not possible with native Win32 threading functions. Thus, MinGW-w64 uses `winthreads` library which provides POSIX threading functions for Windows. However, `winthreads` is not part of the main MinGW-w64 runtime, and it has a different license, which may have to be considered.

Since FB does not care about these C++ features, we can just use MinGW.org toolchains with Win32 threads, and avoid `winthreads`.

- Globbing (command line wildcard expansion etc.) behaviour is different between MinGW.org and MinGW-w64 because they have different runtime libraries/startup code implementations.
 - Globbing is enabled by default in the MinGW.org runtime, MinGW-w64 runtime turns globbing off by default and has `enable-wildcard` configure option. Thus, whether globbing is on or off by default, depends on how MinGW-w64 was built.
 - The way to disable globbing is different:
 - MinGW.org:

```
Extern _CRT_glob Alias "_CRT_glob" As Long  
Dim Shared _CRT_glob As Long = 0
```

- MinGW-w64:

```
Extern _dowildcard Alias "_dowildcard" As Long  
Dim Shared _dowildcard As Long = 0
```

- MinGW-w64 includes DirectX headers needed to compile FB's graphics library. MinGW.org does not contain them; they have to be added manually.
- MinGW.org provides a common installer for their MinGW toolchain in the MSYS shell environment. This makes installing easier than with other toolchains, if MSYS is needed too.

DOS (DJGPP)

FB needs the DJGPP 2.04 beta runtime (does DJGPP 2.03 not work?). In any way, this version of DJGPP is extremely old. On the other hand, there has not been any more recent DJGPP releases, and updates can only be found in DJGPP's CVS. The recommendation is to only use DJGPP CVS if really necessary, though.

Linux

GNU/Linux distros usually provide native gcc + glibc toolchains out-of-the-box and FB is intended to work with them out-of-the-box.

Executables (such as fbc itself) produced on one GNU/Linux distro are not necessarily portable to other GNU/Linux distros, due to differences in system libraries and/or versions, such as glibc version differences, or ncurses/libtinfo differences. The most common problem with fbc is mismatching glibc version, i.e. the fbc binary is run on a system with older glibc than the one it was built with and some form of "glibc too old" error is encountered. The ncurses/libtinfo libraries are always exactly the same either, as shown by the "ospeed" test, which has different results on different systems. Consider re-linking" warnings when running fbc. Also, some distros have separated libncurses and libtinfo, some just have libncurses, which can lead to errors due to the libtinfo shared library not being found.

In theory, it is possible to use static linking to avoid the problems with shared libraries:

- The `fb -static` command line option tells the linker to prefer static libraries instead of shared ones. This can (in theory) also be used when building `fb` itself. It relies on the Linux distro to provide static versions of the system libraries. Linking statically on GNU/Linux is typically discouraged though, in particular with `glibc` (some of its components are not designed for static linking), but also in general (shared libraries are preferred to avoid redundancy).
- `fb` can (in theory) also be used with a different `libc` (instead of `glibc`) that explicitly supports static linking, for example **`musl-libc`**.

In this context, you will typically use a custom `gcc` toolchain, which also means that `fb` has to be built specifically for that toolchain. This approach in general works well, but it can be a lot of work.

Besides that, other `libc`'s may not be ABI-compatible with `glibc`, which can cause problems for `fb` programs if they are written for `glibc`. Most noticeably, the Linux CRT headers are based on `glibc`. An example of an ABI difference between `musl-libc` (0.9) and `glibc` was the `jmp_buf` structure size (used with `setjmp()/longjmp()` functions). As the `fb` CRT headers defined the `glibc` version, they were incompatible to `musl-libc` which used a smaller `jmp_buf` structure.

Another headache when using a different `libc` than the Linux distro default is that you also need to build a lot of libraries such as `ncurses`, `X11` and `Mesa` (and many others) in order to satisfy `fb`'s dependencies, not to mention any other third-party libraries you want to use in your program. Existing libraries precompiled for `glibc` can probably not be used (at least not safely) due to the two `libc`'s being incompatible.

See also

- **`Known problems when compiling fb`**

Compiling the test suite



The FreeBASIC project has a suite of tests which ensure that bugs stay dead and that new bugs have a harder time of gaining a foothold. The test suite is written with the FreeBASIC port of the CUnit library (Thanks stylin!).

Invocation

The tests are located in the tests subdirectory within the main FreeBASIC directory. Invoking with make will present the following help text:

```
$ make
usage: make target [options]

Targets: (using cunit):
cunit-tests
log-tests
failed-tests
check
mostlyclean
clean

Targets: (bypassing cunit)
log-tests ALLOW_CUNIT=1
failed-tests ALLOW_CUNIT=1
mostlyclean ALLOW_CUNIT=1
clean ALLOW_CUNIT=1

Options:
FBC=/path/fbc
FB_LANG=fb | fblite | qb | deprecated
DEBUG=1
EXTRAERR=1
ARCH=arch (default is 486)
OS=DOS
FPU=fpu | sse

Targets: Configuration and Checks
check

Example: make all available tests
make cunit-tests
make log-tests
```

```
Example: make obj -lang qb tests
make log-tests FB_LANG=qb
```

When you make an invocation, such as:

```
make cunit-tests && make log-tests
```

Some initial generation of index files will take place, followed by the compilation of hundreds of tests. Be patient, it can take a while to run all of the tests...

If you get an error message like: FreeBASIC/bin/linux/ld: cannot find libcunit

This means you need to install the cunit library. On Ubuntu this looks like

```
$ sudo apt-get install libcunit1-dev
```

Known Failures

As of the writing of this document, the following tests are expected to fail on some platforms:

```
Suite fbc_tests.string_.format_, Test number format test had failures:
```

1. string/format.bas:168 - CU_ASSERT_EQUAL(sWanted,sResult)
2. string/format.bas:168 - CU_ASSERT_EQUAL(sWanted,sResult)
3. string/format.bas:168 - CU_ASSERT_EQUAL(sWanted,sResult)
4. string/format.bas:168 - CU_ASSERT_EQUAL(sWanted,sResult)

So if you get these failures, everything is normal. No other tests should ever fail, including log tests.

Thank you for running the tests and contributing to make FreeBASIC a healthy compiler! Please report any other failures to <http://www.freebasic.net/forum> so we can investigate.

Glossary - common terms used in fbc development



arg, argument

An expression passed to a parameter in a procedure call.

cast

A type cast changes the compile-time data type of an expression and either causes a conversion (e.g. float <-> int) or a reinterpretation of the expression value's bit representation (e.g. integer <-> uinteger).

comp, compound

- Compound blocks in the language: Any code block that allows nested code such as IF blocks, SCOPE blocks, NAMESPACE blocks, etc. is called a compound.
- Compound symbols: UDTs, sometimes also namespaces, because both may contain nested (namespaced) symbols and they share some common code.

conv, conversion

A conversion is an operation that translates between two different representations of the same value (e.g. float <-> int, or 32bit <-> 64bit).

cast and conv are often used interchangeably in the compiler sources. For example, the AST's CONV nodes represent type casts, no matter whether they perform conversions or not.

Some (but not all) casts require run-time conversions, for example:

short <-> integer

single <-> integer

single <-> double

Simple casts between types of equal class and size do not require a run time conversion, because the bit representation wouldn't change anyways. For example:

short <-> ushort

integer <-> uinteger

These are also called noconv casts.

ctor, constructor

- UDT constructor
- module constructor

ctx, context

UDTs/"classes" in the fbc sources for holding global information shared amongst multiple procedures or modules.

desc, descriptor

- Dynamic string descriptor
- Dynamic array descriptor

dtor, destructor

- UDT destructor
- module destructor

fbc

- The FreeBASIC compiler project as a whole, the Git repository, the project registered on Sourceforge
- The compiler program binary/executable (fbc or fbc.exe), as built from the compiler sources
- The compiler's main module/frontend/driver

fbctinf

FB compile-time information, also see objinfo.

fbgfx

FB graphics, usually referring to the use of FB's built-in graphics keywords, implemented in gfxlib2

frontend stage 1

Compilation of the .bas input files into the next intermediate format: .asn (-gen gas), .c (-gen gcc) or .ll (-gen llvm)

frontend stage 2

Compilation of the .c (-gen gcc) or .ll (-gen llvm) intermediate files into .asm files. (doesn't apply to -gen gas because there the FB compiler

generates .asm itself directly)

function

A procedure with result value; sometimes also used in place of procedure, as in C.

gfxlib2

The FB graphics runtime library implementation from the fbc project.

hashtb

A hash table, often used together with a symbol table to allow fast lookup of the symbols in that symbol table.

libfb, libfbmt, libfbgfx, libfbgfxmt

Names of the libraries built from the rlib/gfxlib2 sources. Libraries name `lib*mt` are the thread-safe versions of their `lib*` counterparts. They are built with the `ENABLE_MT` #define.

local

- Sometimes: A variable allocated on stack
- Any symbol in a nested scope, not the global/toplevel namespace
Scoped static variables also have the `FB_SYMBATTRIB_LOCAL` attribute, even though they are not allocated on stack.

method

- A member-procedure with `THIS` parameter. Static member-procedures (those without the `THIS` parameter) do not have `FB_SYMBATTRIB_METHOD`.
- Sometimes: Any member-procedure, with or without `THIS` parameter

noconv cast

A cast that does not require a conversion.

normal build

Described here: **Normal vs. Standalone**

objinfo

See DevObjinfo

param, parameter

Procedure parameters as declared in procedure DECLARE statements or bodies.

paramvar

For each parameter, the compiler will create a corresponding local variable in the procedure's scope, allowing the parameters to be accessed by user code.

proc, procedure

Any sub or function, including constructors/destructors, operator overloads, property setters/getters.

standalone build

Described here: [Normal vs. Standalone](#)

static

- static variable allocation: on the heap instead of the stack, but still scoped -- also see local.
- static member variables: are actually externs.
- static member procedures: member-procedures without a THIS parameter, also see method.
- "static array" is often used in place of "fixed-size array" (QB language)

struct, structure

TYPE OR UNION, also known as struct/union in C.

sub

A procedure without result (with VOID result).

symtb

A symbol table: owns a linked list of FBSYMBOL in a specific scope. This is where FBSYMBOLS live.

rtlib

The FB runtime library implementation from the fbc project

UDT, user-defined type

TYPEs/UNIONs/ENUMs, sometimes just TYPEs/UNIONs.

vreg

Virtual registers are used when emitting the AST. The AST creates a vreg for the operands and results of all operations that make up the input program. Each backend emits them differently:

- The ASM backend actually maps the vregs to real register and also re-uses them as they become free again. The vregs then also let the x86 code emitter know which exact registers are used.
- The C backend sometimes emits vregs as temporary variables, sometimes simply inserts the expression whose result is represented by a vreg in place of that vreg's first use.
- The LLVM backend simply emits each vreg as a numbered intermediate value.

Since the C/LLVM backends don't re-use vregs, the vregs are almost in static-single-assignment form; although not quite because there still are self-operations etc. produced by the AST which don't take SSA form into account.

In general

Packaging and Manifests

Toolchain/build environment

Release making script

FB manual/documentation

Summary: currently the easiest way to build a release

In general

Making an FB release means:

- Ensuring that the development version is in reasonable/usable state.
- Updating the documentation (Wiki and man page) for language/compiler changes and new features, if not yet done.
- Choosing and preparing gcc toolchains/build environments for DOS, Linux x86, Linux x86_64, Win32, Win64.
- Compiling the development version of FB for all of them.
- Building the Win32 installer (contrib/nsis-installer/).
- Testing the builds to ensure they are basically working.
- Synchronizing the online Wiki with the Wiki files in the fbc Git repository.
- Regenerating the PrintToc and CompilerErrMsg pages.
- Regenerating the examples/manual/ directory (code examples from the Wiki).
- Compiling the offline documentation (CHM, HTML, text).
- Creating the release packages (source code, binary builds, documentation).
- Uploading them and source code of dependencies (binutils, gcc, MinGW, DJGPP, ...) to fbc's download site on SourceForge.
- Announcing the new release on freebasic.net, in freebasic.net/forum News, and in SourceForge fbc project News.

The new release should be compilable with the previous version, so

others can bootstrap it if wanted. Ideally it is compilable with even older versions.

FB releases in form of prebuilt binaries should be made at least for DOS Linux, and Win32. The DOS and Win32 packages traditionally are standalone builds coming with prebuilt binutils and MinGW/DJGPP libraries. The Linux package traditionally is a normal build intended to be installed into `/usr` or `/usr/local` and uses the system's binutils/libraries.

All the binary packages must effectively be built from the same source revision. All the to-be-released fbc binaries should be built with the same date, preferably on the same day the release is published. It's confusing to have multiple fbcs each with the same version number but different dates; are they the same version or not?

The sources must be packaged and uploaded in parallel to the binary packages. That includes sources for third-party binaries included in the FB binary packages, e.g. binutils, gdb, gcc, DJGPP/MinGW libs, etc.

To test the releases, it can be useful to

- run the test suite (for every target system)
- test all compilation modes (exe, dll, profiling, ...)
- run every `.exe` (binutils etc.) included in the packages to ensure that no DLLs are missing
- check that globbing works ok for Windows builds (all included `.exe`'s and new generated ones too), because it might depend on the configuration of the MinGW-w64 runtime.

Linux packages must be `.tar.gz`, Windows/DOS packages must be `.zip`. Other formats such as `.tar.xz` or `.7z` should be offered additionally, but note that there are people with e.g. older GNU/Linux systems that don't know `.tar.lzma` or `.tar.xz`, or with Windows systems that don't have 7-zip installed.

Packaging and Manifests

The FB makefile offers the `gitdist` command for packaging the source code via `git archive`, and the `bindist` command for packaging

previously built binaries. Example workflow:

```
# Go to fbc Git clone
cd fbc

# Compile FB
make

# Package the source code
make gitdist

# Package the binaries, regenerate the manifest
make bindist

# Check the manifest
git diff
```

`gitdist` creates source tarballs in multiple formats. It assumes that all changes to the fbc source code used for building the release have been committed to Git.

`bindist` creates the needed binary archive(s), potentially in multiple formats, with the proper package name and directory layout depending on the target platform and whether it's a normal or standalone build, and it (re)generates the corresponding manifest (list of all files included in the archive) in the `contrib/manifest/` directory in the fbc source tree.

By checking the manifest differences via Git (`git diff`, `git gui`, etc.) you can check whether any files are missing in comparison to the previous release, or whether files were added that should not be included. Should there be any such issues, they may need to be fixed manually (possibly the makefile's `bindist` implementation needs updating, or you simply need to copy in missing files), after which `make bindist` can be run again to recreate the package and update the manifest again.

`bindist` configuration options:

- `TARGET_OS/TARGET_ARCH` makefile variables: You can set `TARGET_OS` and/or `TARGET_ARCH` on the make command line to override the

makefile's default uname check. This is useful if you want to package for a different system than what the uname command returns. For example, packaging the FB-dos release from a MinGW/MSYS shell (with MSYS tools instead of DJGPP tools):

```
make bindist TARGET_OS=dos
```

- FBPACKAGE makefile variable: Package/archive file name without path or extension. Defaults:
 - Linux/BSD normal, Windows/DOS standalone: FreeBASIC-x.xx.x-target
 - Linux/BSD standalone: FreeBASIC-x.xx.x-target-standalone
 - Windows/DOS normal (MinGW/DJGPP-style packages): fbc-x.xx.x-target
- FBPACKSUFFIX makefile variable: Suffix string that will be appended to the package name (and the toplevel directory in the archive).
- FBMANIFEST makefile variable: Manifest file name without path or extension. The defaults are the same as for FBPACKAGE, except without the -x.xx.x version number part.
- FBVERSION makefile variable: Is already set in the makefile, but you can override it if you want to (e.g. when making testing releases instead of "official" releases). For example: FBVERSION=0.90.1 or FBVERSION=0.90.1rc1
- DISABLE_DOCS=1 makefile variable: If this variable is set, bindist will exclude documentation (readme, changelog, man page) and examples from the package. This is useful when creating small binary-only fbc packages such as those for installation into DJGPP/MinGW trees.

Toolchain/build environment

When making an FB release, the GCC toolchain used to build FB has a huge impact, because FB itself will basically become a modified/extended version of that toolchain. The FB-dos and FB-win32 releases include libraries from the used DJGPP/MinGW toolchains, and

they will be used for any FB programs made with those FB builds. Even the FB-linux release will depend on the gcc/glibc version it was built with because of the precompiled rtlib/gfxlib2 libraries, and because of fbc which will have been linked against shared libraries that may not exist on other systems.

Additionally, different GCC toolchains and runtime libraries (e.g. MinGW.org vs. MinGW-w64, or DJGPP 2.03 vs. 2.04 vs. CVS) can be more or less different in terms of ABI compatibility or runtime behaviour. As such any FB program can behave differently depending on the GCC toolchain, including fbc itself.

More information:

Known problems when compiling FB GCC toolchain choice

Release making script

The FB sources contain a release-making script at `contrib/release/build.sh`.

This script downloads & extracts DJGPP/MinGW.org/MinGW-w64 toolchains, FB packages for bootstrapping, fbc sources, etc., then builds normal and standalone versions of fbc, and finally creates the complete packages ready to be released.

- Downloaded archives are cached in the `contrib/release/input/ dir`
- Output packages & manifests are put in the `contrib/release/output/ dir`
- Toolchain source packages are downloaded too
- fbc sources are retrieved from Git; you can specify the exact commit to build, the default is "master".

Usage:

```
cd contrib/release
./build.sh
```

<target> can be one of:

- `dos`: DOS build: must run on Win32. Uses Win32 MSYS, but switches to DJGPP for building FB.
- `linux-x86`, `linux-x86_64`: native builds on GNU/Linux x86/x64_64 relying on the host toolchains; no gcc toolchain is downloaded; no standalone version of FB is built.
- `win32`: 32bit MinGW-w64 build: must run on Win32. Uses MSYS.
- `win32-mingworg`: 32bit MinGW.org build: must run on Win32. Uses MSYS.
- `win64`: 64bit MinGW-w64 build: must run on Win64. Uses Win32 MSYS, but overrides the FB makefile's uname check in order to build for 64bit instead of 32bit.

Requirements:

- MSYS environment on Windows with: bash, wget/curl, zip, unzip, patch, make, findutils (win32/win64 builds need to be able to run `./configure` scripts, to build libffi)
- 7z (7-zip) in the PATH (win32/win64)
- makensis (NSIS) in the PATH (FB-win32 installer)
- git in the PATH
- internet access for downloading input packages and fbc via git

Some of the ideas behind this script:

- Automating the build process for FB releases => less room for mistakes
- Starting from scratch everytime => clean builds
- Specifying the exact DJGPP/MinGW packages to use => reproducible builds
- Only work locally, e.g. don't touch existing DJGPP/MinGW setups on the host

FB manual/documentation

- See also `doc/fbchkdoc/readme.txt` and `doc/manual/readme.txt`

- Get MySQL, libcurl, libaspell, libpcre
- Build the wiki tools:

```
cd doc/libfbdoc
make
cd ../fbdoc
make
cd ../fbchkdoc
make
cd ../makefbhelp
make
```

- Update the wiki cache (the offline copy of the *.wakka files)

```
cd doc/manual
rm -f cache/*
make refresh
```

- Regenerate the PrintToc page:

```
cd doc/fbchkdoc
./mkprntoc -web
```

- Regenerate the CompilerErrMsg page:

```
cd doc/fbchkdoc
./mkerrlst
fbc mkerrtxt.bas -exx
./mkerrtxt > errors.wakka
```

Then copy the error list from errors.wakka into doc/manual/cache/CompilerErrMsg.wakka, and update the online wiki too.

- Update the wiki samples in examples/manual/ (may want to clear out the old ones first, to delete those removed from the wiki)

```
cd doc/fbchkdoc
./getindex -web
./samps extract @PageIndex.txt
```

Summary: currently the easiest way to build a release

- Update the wiki snapshot in the fbc sources
- Regenerate PrintToc and CompilerErrMsg
- If needed, update wiki samples in examples/manual/
- Build documentation packages (CHM on Windows, rest can be done on Linux)
- Check whether toolchains used in the contrib/release/build.sh script need updating

- Have target systems ready (installations of Linux and Windows, 32bit and 64bit -- virtual machines are useful for this)
- For each system, update fbc sources (to have the latest version of the release script)
- On win32:

```
cd contrib/release
./build.sh win32
./build.sh win32-mingworg
./build.sh dos
```

- On win64:

```
cd contrib/release
./build.sh win64
```

- On linux-x86:

```
cd contrib/release
./build.sh linux-x86
```

- On linux-x86_64:

```
cd contrib/release
./build.sh linux-x86_64
```

- Collect all the archives and manifests from the contrib/release/input and contrib/release/output directories
- Review the manifests to check for missing files etc.
- If ok, commit the new manifests
- Create the release tag
- Upload the packages
- Post announcements

Bootstrapping fbc on a new system



fbc is written in FB itself, so you need a working fbc to build a new fbc. F to the target system, or full cross-compiling using a gcc cross-compiler t

Bootstrapping using the FreeBASIC-x.xx.x-source-bootstrap packa

The FreeBASIC-x.xx.x-source-bootstrap package contains the FB sourc

```
make bootstrap
```

(as long as the package contains the precompiled sources for the target

This package can be created by running:

```
make bootstrap-dist
```

Doing `make bootstrap-dist`, taking the package to the target system, an

Bootstrapping by precompiling the compiler sources

- On Linux or Win32 (or another system where you have a working

```
fbc -e -m fbc src/compiler/*.bas -r -target -arch
```

Some random examples:

```
x86 Win32 -> x86 OpenBSD: -target openbsd [-arch 486]
```

```
x86 Win32 -> x86_64 FreeBSD: -target freebsd -arch x86_64
```

```
x86 Linux -> ARM Linux: -target arm-linux-gnueabi, or just -ar
```

- On the target system, compile FB's `rtlib/gfxlib2` using the native C

```
make rtlib gfxlib2
```

- Take the .asm or .c files (produced in the first step) to the target s
 - If you produced .asm files, take them to the target system,

```
for i in src/compiler/*.asm; do
as $i -o `echo $i | sed -e 's/asm$/o/g'`
done

gcc -o fbc lib/freebasic//fbrt0.o src/compiler/*.o -Llib/freebas:
```

- If you produced .c files, take them to the target system, ar

```
gcc -o fbc -nostdinc -Wall -Wno-unused-label -Wno-unused-function
```

Additional notes & tips

- The new fbc and the new rtplib/gfxlib2 must be built from the same
- When linking fbc for a Unix-like system, you need to link it agains
- An alternative to linking with gcc is to invoke ld manually, like fbc

Bootstrapping by cross-compiling everything

If you're on Linux or Win32 or another system where you already have a cross-compile an FB setup like so:

- Build a native FB setup with additional libraries for cross-compilin

```
# Get a directory with the fbc sources, e.g. "fbc"
cd fbc
make
make rtplib gfxlib2 TARGET=

# Optionally, you can install everything into /usr/local:
make install
make install-rtlib install-gfxlib2 TARGET=
```

- Use the native FB setup built above to cross-compile the new FB

```
cd ..
mkdir crosscompiled-fbc && cd crosscompiled-fbc
make -f ../fbc/makefile FBC='../fbc/bin/fbc -i ../fbc/inc' TARGET='x86_64-w64-mingw32'
# (Specifying FBC=... is only needed if you did not install it g
```

Cross-compiling the 64bit version on a 32bit system with gcc -m64

If you have a gcc multilib toolchain with -m64 support on a 32bit system, the MinGW-w64 project also have support for cross-compiling to 64bit vi

```
# Get FB sources into fbc/ (must be 0.91+ because earlier version
# and build a native (32bit) FB first
cd fbc
make

# Then add the 64bit rtlib/gfxlib2 to that. Specifying MULTILIB=64
make rtlib gfxlib2 MULTILIB=64

# Now we have a new 32bit FB with 64bit libraries for cross-compiling
# This can now be used to build a full 64bit FB:
cd ..
mkdir fbc64
cd fbc64
make -f ../fbc/makefile MULTILIB=64 FBC='../fbc/bin/fbc -i ../fbc/inc'
```

This does not only work with gcc -m64 on 32bit, but also with gcc -m32 on

Under Construction

About This Guide

This guide is not a **C tutorial** or a **step by step guide** for converting headers. This is a style guide which represents the `ideal` header we would like to maintain. Currently not all of the headers under our control conform to this guide 100%, but work is in progress to do this and all new contributions should attempt to use these standards.

General

- Translations should be very close to the original, so they look familiar and can be updated easily.
- Identifiers (including any `#defines`) should not be changed unless absolutely necessary.
- Smaller files may be combined into one bigger header, if they would be `#included` anyways and all belong to the same library.
- Original license should be retained.

Coding style

- Headers need to work with the latest FreeBASIC version.
- Naming conflicts between multiple identifiers (due to FreeBASIC's case insensitivity) or an identifier and a FreeBASIC keyword should be resolved by appending an underscore to one identifier.
- `extern "c"` blocks should be used instead of `cdecl` alias `"..."` for function declarations or function pointer types.
- Preprocessor directives (including `#defines`) should be preserved. Exception: Remove if they serve only to select options for different C compilers, i.e. `extern` differences, then these can be removed unless they provide support for further code. When choosing compilers the choice should favor GNU C.
- FreeBASIC keywords should be lower-case.

Dealing with constructs not supported by FreeBASIC

- Inline functions should be converted to a macro if appropriate.
- Preprocessor directives inside structure declarations, function bodies, or similar may need to be moved outside because in FreeBASIC they'd be scoped.
- Declarations spread across multiple lines with preprocessor directives in between them (for example function declarations, or array initializers) will need to be manually rewritten

Quick overview of all modules



(Only somewhat sorted)

fbc

Frontend: main module, entry point, command-line handling, assembling/linking/etc.

objinfo

Object/library information section reader/writer, used by fbc. Includes tiny ELF/COFF object file format readers.

fb

FB parser interface, starts the parser for every input/include file.

parser

Recursive parser, asks lex for tokens, builds up the ast.

lex, pp

Lexer/tokenizer and preprocessor directive parsing.

error

Error reporting functions, used by many parts of fbc, mostly the parser though.

rtl

Helper functions to build up the ast nodes for rtplib/gfxlib function calls. Declarations must match the actual functions in the rtplib/gfxlib2 source code.

symb

Symbols lookup and storage (information on variables/functions), scope/namespace handling, name mangling; used by parser/ast/emitter

ast

Abstract syntax tree: per-function code-flow + expressions.

astNew*(): Node creation/tree building, used by the parser.

astLoad*(): First step in emitting, calls ir, called after each function is parsed.

ir, ir-hlc, ir-llvm, ir-tac

Intermediate representation interface (using virtual registers) used to emit the ast.

hlc: High level C emitter (high level in comparison to the ASM backend anyways)

llvm: LLVM IR emitter

tac: Three-address-codes module (asm backend), calls emit. Responsible for register allocation, reusing, spilling.

reg

Register allocator for ir-tac.

emit, emit_SSE, emit_x86

Assembler emitter abstraction and SSE/x86 emitters.

edbg_stab

Stabs debug format emitting for emit_x86.

dstr

Dynamic z/wstrings, used mostly by lex.

hash

Generic hash table, used by symb/fbc.

hlp, hlp-str

Helper functions for all parts of the compiler, plus another implementation of dynamic z/wstrings.

list

Generic linked list with built-in memory pool, used a lot. This is often used as pure pooled allocator, for example for AST nodes or symbols.

flist

list-based without deletions.

pool

list-based allocator using multiple lists with node sizes ranging from small to large, allowing it to store away strings into the next best fitting chunk

waste as less memory as possible. Used to store away symbol identifiers.

stack

Generic list-based stack.

fbcc stores extra information into the object files (.o) it generates, in order to read it out again at link-time. The information that is stored currently consists of the -lang/-mt settings and all libraries/search paths (-l, #inclib, -p, #libpath) that were specified when compiling that object file. This way fbcc can show a warning when mixing object files that were compiled with different options, because they may be incompatible, and fbcc can automatically link in libraries that were specified via #inclib, even if the user compiles and links in separate steps.

This is accomplished by emitting an extra section called "fbctinf" (FreeBASIC compile time information?) when compiling, and reading it back in at link-time. Furthermore, when building a static library, fbcc creates an extra object file (called __fb_ct.inf) containing just that extra information and adds it to the library. At link-time fbcc looks at each library to figure out whether it has such an __fb_ct.inf file or not.

In order to do this fbcc has a custom COFF, ELF32 and also archive file format readers that can extract the .fbctinf section content. Previously, fbcc used libbfd from binutils to do this, however depending on libbfd is problematic especially because of its highly unstable ABI.

Memory management



fbcs tries to avoid memory allocations as much as possible, since they are pretty slow generally. The linked list implemented in list.bas comes with a builtin memory pool, so pretty much every list is pooled. The memory pool pre-allocates large chunks and can then quickly hand out many small nodes. Those lists are used for simple things like the list of libraries to link into an executable, but also for heavier things like AST nodes. The memory pool is supposed to speed things up (no idea if this was ever verified though).

In many places the compiler simply uses global/static variables, for example fixed-length strings, in order to avoid memory allocations. Tokens are a nice example: lex.bas parses input characters into tokens, and stores the token text in static buffers. Token text, that could be: variable names, string literals, and so on. All tokens are stored here though, so the preprocessor can correctly record macros. Now take into account the huge number of tokens the parser has to deal with: For example, FB's current Windows headers result in ~100k tokens. Dynamically allocating a buffer for every token would quickly become inefficient.

Of course the token length is limited by using a static buffer, but fbcs's default of 1024 bytes should be enough for everyone. Similar length limitations apply to many things in the compiler because of the use of fixed-length buffers. In most situations, the buffers in the compiler are not used to their full potential, i.e. they are bigger than they need to be.

All that does not mean the compiler does not use dynamic memory allocations at all. It does, in situations when allocating is easier than using a list/pool and speed is not critical. FB's builtin string type is used in many places too. As long as the string's are kept allocated, they are very efficient. Expansion of macro parameter stringifying in the pre-processor uses a strReplace() based on string's, and it is fast (enough). Besides that, dynamic strings, which are basically the same as string's, are used everywhere in the pre-processor, from macro recording to macro expansion.

Out-of-memory situations/allocation failures are not seriously handled. There are NULL checks in some places where `allocate()` is called, but these checks are pointless, since the rest of fbc does not check for NULL. NULL is sometimes used to indicate an error, for example by some `astNew*()` functions. Also, the compiler does not `deallocate()` everything, but lets the OS do the cleanup.

Lexer & preprocessor



lex*.bas: File input, tokenization, macro expansion buffer, token queue,
pp*.bas: Preprocessor directive parsing, macro expansion text construct

The lexer reads the source code from the .bas files and translates it into
sees this:

```
dim as integer i = 5
print i
```

as:

```
(Top-level parser retrieves the first token:)
DIM      keyword      (Go to variable declaration parser)
AS       keyword      (Go to datatype parser)
INTEGER  keyword      (Data type)
"i"      symbol       (Back to variable declaration, variable :
"="      operator     (Go to initializer parser)
"5"      number literal (Expression)
EOL      statement end (Variable declaration parser is done,
                        the variable is added to the AS
                        back to toplevel parser)

(Next line, next statement)
PRINT    keyword      (Go to QB print quirk function call parse
"i"      symbol       (Expression, lookup "i" symbol, it's an :
                        create a CALL to fb_PrintInt(), the
EOL      (Print parser is done, back to toplevel)
EOF      (Top-level parser is done)
```

The lexer is an abstraction hiding the ugly details of user input (indentation, #includes) from the parser. Additionally it does preprocessing, consisting of preprocessor directive parsing. The general idea is to handle all preprocessing so the parser never calls preprocessor functions, the lexer

Tokens

Macro storage and expansion

Preprocessor directive parsing
File contexts
Quick overview of the call graph

Purpose



`fb.bas`: Main module for the compiler, parent module for parser/lexer/AST/IR/emitters, toplevel file & include file handling
`parser*.bas`: Parsing/compilation functions: lexer tokens -> AST nodes.
`symb*.bas`: Symbol tables and lookup, namespace/scope handling.
`rtl*.bas`: Helpers to build AST calls to `rtlib/gfxlib` functions.

The structure of the parser has a very close relation to the **FreeBASIC grammar**. Basically there is a parsing function for every element of the grammar.

The parser retrieves tokens from the lexer and validates the input source code. Most error messages (besides command line and file access errors) come from here. Additionally the parser functions build up the corresponding AST. This is the heart of the compilation process.

Many of the parser's (or rather compiler's) functions (prefixed with a 'c') parse and skip the grammar element they represent, or show an error if they don't find it. The parser is fairly recursive, mostly because of the expression parser and the `#include` parsing.

From parsing to emitting

When parsing code a corresponding AST is built up to represent the program. The AST is used to represent executable code, but also to hold temporary expressions, for example the values of constants or the initializers found while parsing type or procedure declarations. The AST does *not* contain nodes for code flow constructs like IF, DO/LOOP, GOTO, RETURN, EXIT DO, etc., but it contains labels and branches. Likewise, several operations (like `IIF()`, `ANDALSO`, `ORELSE`, field dereference, member access) are replaced by the corresponding set of lower-level operations in the AST.

After parsing a function, the AST for this function is optimized, and then emitted recursively via `astLoad*()` calls on each node, from the top down. Note that each AST node has its own implementation of `astLoad()`.

Top level parsing process



`fb.bas:fbCompile()` is called from the `fb` frontend for every input file. Parsing (and compiling) of the file begins here.

`fb.bas:fbCompile()`

- Open the input `.bas`
- Start the emitter (`ir`) (Open the output `.asm`)
- `fbMainBegin()` (Build the AST for the implicit `main()` or static constructor for module-level code)
- `fbPreIncludes()`
 - `fbIncludeFile()` for every preinclude (found on the `fb` command line)
- `cProgram()`
- `fbMainEnd()` (Close the implicit `main()`)
- Finish emitting (`ir`) (Finish generating the `.asm` and close it)
- Close the input `.bas`

`fb.bas:fbIncludeFile()`

- Include file search
- `lexPush()` (Push a new lexer context to parse this `#include` file without disturbing the lexer's state in the parent file)
- Open the include file
- `cProgram()`
- Close the include file
- `lexPop()` (Restore the lexer state to the parent file)

`parser-toplevel.bas:cProgram()` is the root of the **FB grammar**, and parses a file. Here's a short & quick run down of what is done:

- `cLine()` repeatedly until EOF
 - `cLabel()`
 - `cStatement()`
 - Declarations
 - UDT declarations, typedefs

- Variables (DIM, VAR, ...)
- Procedure declarations (DECLARE)
- Procedure bodies (SUB, FUNCTION, ...)

(Procs temporarily replace the implicit module level procedure, so any AST nodes go into them instead of the implicit main())

- Compound statements (IF/ELSE, DO/LOOP, EXIT/CONTINUE DO, ...)
- Procedure calls
- Function result assignments
- Quirk statements (special QB rtlib/gfxlib statements)
- ASM blocks
- Assignments
- Procedure pointer calls

and most of them use cExpression() at some point.

Symbols



In order to be able to make the transition from tokens to AST, the parser needs to be able to recognize functions, variables, types, etc. The `symp` module keeps track of all these symbols and their namespaces and scopes. The parser can do lookups in the current scope, or in just specific namespaces. Many AST nodes have a corresponding symbol (e.g. variables and functions).

Representation of data types



Almost all parts of the compiler deal with data types in one way or another. The most common is what most of the compile-time type checks are based on: basic data types, that takes care of expressions (including casting/conversions).

A data type is represented as a combination of:

- dtype integer
 - 5 bits: raw type:
 - void (unknown type, e.g.: any ptr, type t as t)
 - byte, ubyte
 - char (zstring pointers and their deref expressions)
 - short, ushort
 - wchar (wstring pointers and their deref expressions)
 - integer, uinteger
 - enum (integer)
 - long, ulong
 - longint, ulongint
 - single, double
 - string (variable length)
 - fixstr (fixed length strings, string * N, N is the length)
 - struct (UDT, -> subtype is used)
 - namespace (used during name mangling?)
 - function (used for function pointers, -> subtype is used)
 - forward reference (will be changed to actual type when subtype is used)
 - pointer (this value is only used temporarily as a macro)
 - xmmword (used by SSE emitter)
 - 4 bits: PTR count

How many PTR's there are on the type, maximum 8. If > 0, then the data

- 9 bits: CONST mask (8 PTR's + 1 "base")

Example	CONST mask	
const integer	000000001	(first CONST
integer const ptr	000000001	(ditto)
const integer ptr	000000010	(pointer to c
const integer ptr const ptr	000000101	(const pointer

- subtype, which for some types points to symbol:
 - For UDTs types (structs/classes, enums) this points to symbol
 - For forward-referencing typedefs this points to a symbol which will eventually be replaced by the actual subtype
 - For procedure pointers, this points to an anonymous calling convention etc. and most importantly the type
- length integer

This is used in places that have to calculate sizes (e.g. structure size calculation stack offsets).

Select Case



Basic implementation

```
dim i as integer                                dim i as integer
                                                scope
select case i + 123                            dim temp as integer = any
                                                temp = i + 123
case 1                                          if( temp
1 ) then goto cmplabel1
    print "1"
case 2                                          scope
                                                print "1"
                                                end scope
                                                goto endlabel
2 ) then goto cmplabel2
    print "2"
case else                                       cmplabel1:
    print "else"                               if( temp
                                                scope
                                                print "2"
                                                end scope
                                                goto endlabel
end select                                       cmplabel2:
                                                scope
                                                print "else"
                                                end scope
                                                cmplabel3:    ' ' unused on
                                                endlabel:
end scope
```

- SELECT CASE
 - opens the implicit outer scope
 - declares the temp var
 - when inside a procedure with STATIC, the temp var
 - the FB_SYMBATTRIB_TEMP is removed from the

- emits the assignment
 - declares the end label
- each CASE
 - if there was a previous CASE
 - closes the previous CASE's scope
 - emits a jump to the end label
 - emits the label for this CASE
 - emits a conditional branch that jumps to the next CASE if
 - opens the CASE's scope
 - CASE ELSE does not emit a conditional branch
 - once CASE ELSE was used, no further CASE blocks are
- END SELECT
 - closes the previous CASE's scope
 - emits an extra CASE label at the end (There is no CASE c
it is a conditional CASE. The last CASE could jump to the
case handling code.)
 - emits the end label
- any EXIT SELECTs jump immediately to the end label

SELECT CASE on strings/zstrings/fixstrs

<pre> dim s as string select case s + "1" case "1" 0) then goto cmlabel1 print "1" </pre>	<pre> dim s as string scope dim temp as string fb_StrAssign(temp, s) fb_StrConcatAssign(temp, if(fb_StrCompare(temp, scope print "1" end scope </pre>
--	--

```

                                goto endlabel
                                cmlabel1:
                                endlabel:
                                fb_StrDelete( temp )
end select
                                end scope
                                fb_StrDelete( s )

```

- SELECT CASE on string/zstring/fixstr expressions uses a string temp var
 - probably because that's easiest
 - knowing the string length will potentially speed up the following
 - the dynamic memory allocation can be a slow down too
- the string temp var is destroyed at scope end or scope breaks (e.g. block)

SELECT CASE on wstrings

<pre> dim w as wstring * 10 select case w + wstr("1") case wstr("1") </pre>	<pre> dim w as wstring * 10 scope dim temp as wstring dim tempexpr as wstring temp = fb_WstrAlloc(len(w) + 1) fb_WstrAssign(temp, w + wstr("1")) if(fb_WstrCompare(temp, wstr("1")) == 0) then goto cmlabel1 print "1" end scope goto endlabel cmlabel1: endlabel: fb_WstrDelete(temp) end scope </pre>
---	--

- similar to SELECT CASE on zstrings, for wstring expressions a w
- the temp wstring is treated much like a dynamic wstring object w
 - it is a VAR symbol with type WCHAR PTR
 - marked with FB_SYMBSTATS_WSTRING
 - this allows ctor/dtor checks to recognize it and give it the r
- this way, the temp wstring is destroyed at scope end or scope bre

SELECT CASE without temp var

When the expression given to the `select` statement is just a simple variable, the given variable itself will be used in the comparisons at each case.

<pre> dim i as integer select case i case 1 1) then goto cmlabel1 print "1" end select </pre>	<pre> dim i as integer scope if(i scope print "1" end scope goto endlabel cmlabel1: endlabel: end scope </pre>
---	---

Basics

Using FB's built-in functionality, there are four ways of getting keyboard input:

- **Inkey()** returns a string containing an ASCII char corresponding to the key pressed by the user, or a 2-byte FB extended keycode for some special keys, such as the Arrow keys or Page Up/Down. It works pretty much like it did in QB.
- **Getkey()** returns the same information as `inkey()`, but in form of an integer instead of a string. `inkey()` and `getkey()` belong together: They use the same code and they are located in the same modules.
- **Multikey()** takes an **FB scancode (SC_*)** and checks whether that key is pressed at this moment.
- **Screenevent()** returns key presses in form of `EVENT_KEY_PRESS` events (and others for key release or repeat). It returns the **FB scancode** in the **Event.Scancode Field**, and the ASCII char value or 0 in the `EVENT.ascii` field. `EVENT.ascii` does not use FB extended keycodes; the `EVENT.scancode` field can be checked instead in order to handle extended keys.

"**scancode**" refers to the `SC_* #defines` which are more or less matching the DOS keyboard scancodes. The values are not made up, they themselves correspond to certain ASCII chars, for example: `SC_HOME` `asc("G") = &h47.`; They're also the same values that you get under DOS/DJGPP or from the Linux kernel as part of extended key code sequences. Besides their use in `multikey()` or `screenevent()`, scancodes are used in various places internally, for example when translating between different kinds of key codes, as an easy-to-use and portable representation of keycodes.

"key" refers to an ASCII char, or a 2-byte extended keycode string for other keys as returned by `inkey()`. The `rtlib` has several `KEY_* #defines`

for the available 2-byte extended keycodes, in form of integers. These are used internally and also match the values returned by `getkey()`.

FB's 2-byte extended keycodes consist of a `&hFF;` byte followed by a byte containing the `SC_*` scancode value corresponding to the keypress

Checking for `SC_HOME` returned by `inkey()` could look like:

```
if( inkey( ) = chr( 255 ) + "G" ) then ...
```

Checking for `SC_HOME` returned by `getkey()`:

```
if( getkey() = &h47FF; ) then ...
```

```
if( getkey() = ((SC_HOME shl 8) or &hFF;) ) then ...
```

`inkey()`, `getkey()` and `multikey()` use wrapper functions that call ...

- the console-mode versions `fb_ConsoleInkey()`, `fb_ConsoleGetkey()`, `fb_ConsoleMultikey()` by default,
- or the `gfxlib` versions `fb_GfxInkey()`, `fb_GfxGetkey()`, `fb_GfxMultikey()` if a graphics `SCREEN` is active,

by using function pointer hooks.

rtlib

The `rtlib` has separate console-mode implementations of the above functions, for each platform:

- **DOS**

`fb_ConsoleInkey()` and `fb_ConsoleGetkey()` use DJGPP's `getch()` function to retrieve input characters anytime they're called. `getch()` returns ASCII chars, but also 2-byte sequences for special keys, which are easy to handle because they match the `SC_*` scancodes.

`fb_ConsoleMultikey()` installs an interrupt handler that uses port I/O to read keyboard information and updates a key state table which is checked by `multikey()`.

- **Win32**

`fb_ConsoleInkey()` and `fb_ConsoleGetkey()` (indirectly) use the Win32 API functions `PeekConsoleInput()` and `ReadConsoleInput()` to get queued key press/release events whenever needed. All currently pending events are handled during a call, and after very complex internal translation involving `MapVirtualKey()`, the keys are put into a buffer, from where `fb_ConsoleInkey()` and `fb_ConsoleGetkey()` read the keys they return.

`SetConsoleCtrlHandler()` is used to listen for console close/system shutdown events to provide `SC_CLOSE` events for console-mode (the win32 port of the rlib might be the only one going this far).

`fb_ConsoleMultikey()` uses a `FindWindow()/GetForegroundWindow()` hack to determine whether the console window is focused, and if yes, simply uses `GetAsyncKeyState()`.

- Linux, *BSD

The Unix port of the rlib runs a console keyboard handler (and a console mouse handler) in a background thread, in order to provide input for `multikey()` (and `getmouse()`).

`fb_ConsoleInkey()` and `fb_ConsoleGetkey()` read input bytes through the `__fb_con.keyboard_getch()` hook. By default, `__fb_con.keyboard_getch()` points to a simple function that just uses `fgetc()` on `/dev/tty` (indirectly; the Unix rlib initialization code opens the handle, and changes I/O settings etc., not only for the purpose of keyboard input, but mostly).

The terminal returns ASCII chars for simple key presses, and special escape sequences for extended keys. On the first call, various termcap lookups (via `tgetstr()`) are done to determine these terminal-specific escape sequences for certain key press events, and they are put into a lookup tree to allow easy & fast translation to the corresponding FB extended keycodes. By doing the termcap query the Unix rlib can support all the different terminals (e.g. `xterm` vs. `linux`) quite well, although there still are some keys not working here and there.

Only one "event" (ASCII char or escape sequence) is read at a time, the resulting key is added to a key buffer, from where `fb_ConsoleInkey()` and `fb_ConsoleGetkey()` can read it.

`fb_ConsoleMultikey()` is currently implemented for the Linux port only, not under *BSD though. In console-input mode (used under 'console'/linux terminals), it `dup()`licates the rlib's `/dev/tty` handle, and switches it over into medium raw mode. Then it overrides the background thread's `__fb_con.keyboard_handler()` hook to a function that `read()`s kernel key codes from the duplicated `/dev/tty` handle.

Called from the background thread, it reads a fixed amount of input at

once, whenever it arrives. After somewhat complex translation, a key state table is updated to reflect the state of pressed/released keys, to be checked by `fb_ConsoleMultikey()` at any time, and the keys are added to a key buffer from where an overridden `__fb_con.keyboard_getch()` reads them, whenever called by `fb_ConsoleInkey()` or `fb_ConsoleGetkey()` [wh is this done?]. Furthermore, the keys are sent to the Linux fbdev `gfxlib2` driver, if it's active.

In X11 mode (used under 'xterm' terminal), `fb_ConsoleMultikey()` sets the background thread's `__fb_con.keyboard_handler()` to a function that checks whether the xterm has input focus (`XGetInputFocus()`) and if yes, simply uses `XQueryKeymap()` to update the key state table for `fb_ConsoleMultikey()`.

gfxlib2

In the `gfxlib`, `fb_GfxInkey()` and `fb_GfxGetkey()` use one key buffer (same code on all platforms), to which the different/platform-specific `gfx` drivers post keys to. Similar to that, there is a single key state table for `fb_GfxMultikey()`, and it is also updated by the `gfx` drivers. Whether or not the `gfx` drivers actually do post keys or update key states is up to them though.

- **DOS**

The DOS `gfxlib2` port (for all DOS `gfx` drivers) sets a hook/callback that's called by the same keyboard interrupt handler used by the DOS `fb_ConsoleMultikey()`.

- **Win32 driver**

The `gfx` window thread listens to `WM_KEYDOWN`, `WM_CHAR` and `WM_CLOSE`, translates the keys, and then updates the key state table, posts them to the `fb_GfxInkey()/fb_GfxGetkey()` buffer, and fills in & posts the corresponding `EVENT` for `screenevent()`.

- **X11 driver**

The `gfx` window thread listens to `KeyPress` and other `XEvent`'s, translates the keys, then posts them etc., just like the Win32 driver.

- **Linux fbdev driver**

As mentioned above, the fbdev driver gets its input from the same keyboard handler code that's used by the Linux `fb_ConsoleMultikey()`.

GNU GENERAL PUBLIC LICENSE



Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
- You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
- If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you

distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

- Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute s as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE

PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

GNU LESSER GENERAL PUBLIC LICENSE



Version 2.1, February 1999

Copyright (C) 1991, 1999 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public License, version 2, hence the version number 2.1.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software packages--typically libraries--of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in a particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for the service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute

copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more

lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

**GNU LESSER GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND
MODIFICATION**

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each license is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) The modified work must itself be a software library.

b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but

is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this

License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

c) Accompany the work with a written offer, valid for at least three years to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either

source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless the component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute s as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain

countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free program whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH

YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the library's name and a brief idea of what it does.>
Copyright (C) <year> <name of author>

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Also add information on how to contact you by electronic and paper mail

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the library `Frot (a library for tweaking knobs) written by James Random Hacker.

<signature of Ty Coon>, 1 April 1990
Ty Coon, President of Vice

That's all there is to it!

GNU Free Documentation License



Version 1.2, November 2002

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc.
51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world wide, royalty-free license, unlimited in duration, to use that work under

the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup,

has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when

you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- * A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- * B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- * C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- * D. Preserve all the copyright notices of the Document.
- * E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- * F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- * G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- * H. Include an unaltered copy of this License.

* I. Preserve the section Entitled "History", Preserve its Title, and add to an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

* J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

* K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

* L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

* M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

* N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

* O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a

passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from the copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provide

that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Interface

The basic public interface of the lexer is from `lex.bas`:

- `lexGetToken()`: Retrieve current token's id, an `FB_TK_*` value.
- `lexGetLookAhead(N)`: Look ahead N tokens
- `lexSkipToken()`: Go to next token
- `lexGetText()`: Returns a zstring ptr to the text of the current token, e.g. string/number literals (their values are retrieved like this), or the text representation of other tokens (e.g. operators).
- some more `lexGet*()` accessors to data of the current token
- `lexPeekLine()`: Used by error reporting to retrieve the current line of code.

Current token + look ahead tokens

Tokens are a pretty short-living thing. There only is the current token and a few look ahead tokens in the token queue. That's all the parser needs to decipher FB code. The usual pattern is to check the current token, decide what to do next based on what it is, then skip it and move on. Backward movement is not possible. The file name, line number and token position shown during error reporting also comes from the current lexer state.

The token queue is a static array of tokens, containing space for the current token plus the few look ahead tokens. The token structures contain fairly huge (static) buffers for token text. Each token has a pointer to the next one, so they form a circular list. This is a cheap way to move forward and skip tokens, without having to take care of an array index. Copying around the tokens themselves is out of question, because of the huge text buffers. The "head" points to the current token; the next "k" tokens are look ahead tokens; the rest is unused. When skipping we simply do "head = head->next". Unless the new head already contains a

token (from some look ahead done before), we load a new token into the new current token struct (via `lexNextToken()`). Look ahead works by loading the following tokens in the queue (but without skipping the current one).

Tokenization

`lex.bas:lexNextToken()`

The lexer breaks down the file input into tokens. A token conceptually is an identifier, a keyword, a string literal, a number literal, an operator, EO or EOF, or other characters like parentheses and commas. Each token has a unique value assigned to it that the parser will use to identify it, instead of doing string comparisons (which would be too slow).

`lexNextToken()` uses the current char, and if needed also the look ahead char, to parse the input. Number and string literals are handled here too. Alphanumeric identifiers are looked up in the `symb` hash table, which will tell whether it's a keyword, a macro, or another FB symbol (type, procedure, variable, ...).

Identifiers containing dots (QB compatibility) and identifier type suffixes (as in `stringvar$`) are handled here too (but not namespace/structure member access). Tokens can have a data type associated with them. That is also used with number literals, which can have type suffixes (as in `&hFFFFFFFFFFFFFFFFull;`).

Side note on single-line comments

Quite unusual, single-line comments are handled by the parser instead of being skipped in the lexer. This is done so that usage of `REM` can easily be restricted as in QB, after all `REM` is more like a statement than a comment. Besides that, comments can contain QB meta statements, so comments cannot just be ignored. Note that the parser will still skip the rest of a comment (without tokenizing it), if it does not find a QB meta statement.

(Multi-line comments are completely handled during tokenization though)

File input

`lex.bas:hReadChar()`

The input file is opened in `fb.bas:fbCompile()`; the file number is stored in the global `env` context (similar for `#includes` in `fb.bas:fbIncludeFile()`). The lexer uses the file number from the `env` context to read input from. It has a static `zstring` buffer that is used to stream the file contents (instead of reading character per character), and for Unicode input, the lexer uses a `wstring` buffer and decodes UTF32 or UTF8 to UTF16. The lexer advances through the chars in the buffer and then reads in the next chunk from the file. EOF is represented by returning a NULL character.

Some terms used in the source code (Note the double meanings):

- `macro`: The `#defined/#macroed` object that will be expanded to its replacement text
- `macro`: a function-like macro, e.g. `#define m(a, b)`
- `define`: an object-like macro, e.g. `#define simple`
- `argless define` (should be called `parameter-less`): a function-like macro without parameters, e.g. `#define f()`

How macros are stored

Macros are basically stored as raw text, not as token runs (as in GCC's `libcpp` for example). The body of simple `#defines` without parameters is stored as one string. Macros with parameters are stored as sequence of "macro tokens". There are three types of macro tokens:

- `text("<text>")`

Raw text, but spaces and empty lines trimmed (like in a `#define` without parameters)

- `textw("<wstring text>")`

Same as above, just for Unicode input.

- `parameter(index)`

A macro parameter was used here in the declaration. The index specifies which one. During expansion, the text of `argument(index)` is inserted where the parameter was in the declaration.

- `stringify_parameter(index)`

Same as above, except the argument will be stringified during expansion

Note: macro tokens are actually `sym.bi:FB_DEFTOK` structures, and they contain an `id` field holding on of the `FB_DEFTOK_TYPE_*` values to tell what they contain.

For example:

```
#define add(x, y) x + y
```

becomes:

```
parameter(0), text(" + "), parameter(1)
```

And the expansion text will be:

```
argument(0) + " + " + argument(1)
```

Storing macros as text is a fairly easy implementation, but it requires to re-parse the macro body over and over again. For example, since GCC works with preprocessing tokens and tokenruns, macros are stored as tokens, making expansion very fast, because there is no need to tokenize the macro body again and again. fbc's implementation is not as flexible and maybe not as efficient, but is less complex (regarding code and memory management) and has an upside too: Implementation of ## (PP token merge) is trivial. ## simply is omitted while recording the macro's body, where as in token runs the tokens need to be merged explicitly.

When are macros expanded?

Because of token look ahead, macros must be expanded during tokenization, otherwise the wrong tokens might be loaded into the token queue. After all the parser should only get to see the final tokens, even during look ahead.

In `lexNextToken()`, each alphanumeric identifier is looked up in the symk module to check whether it is a keyword or a macro. Macros and keywords are kept in the same hash table. Note that macros cannot have the name of keywords; `"#define integer"` causes an error. If a macro is detected, it is immediately expanded, a process also called "loading" the macro (`pp-define.bas:ppDefineLoad()`).

Macro call parsing

If the macro takes arguments, the macro "call" must be parsed, much like a function call, syntax-wise. Since macro expansion already happens in `lexNextToken()`, the source of tokens, the parsing here is a little tricky. Forward movement is only possible by replacing (and losing) the current token. The token queue and token look ahead cannot be relied upon. Instead it can only replace the current token to move forward while parsing the macro's arguments.

Since `lexNextToken()` is used to parse the arguments, macros in the arguments themselves are recursively macro-expanded while the arguments are being parsed and recorded in text form. The argument texts are stored for use during the expansion.

So, a macro's arguments are expanded before that macro itself is expanded, which could be seen as both good and bad feature:

```
#define stringify(s) #s
stringify(__LINE__)
```

results in `2` in FB, but `__LINE__` in C, because in C, macro parameters are not expanded when used with `#` or `##`. In C, two macros have to be used to get the `2`:

```
#define stringize(s) #s
#define stringify(s) stringize(s)
stringify(__LINE__)
```

Putting together the macro expansion text

The expansion text is a string build up from the macro's body tokens. For macro parameters, the argument text is retrieved from the argument array created by the macro call parser, using the indices stored in the parameter tokens. Parameter stringification is done here.

There is a specialty for the builtin defines (`__LINE__`, `__FUNCTION__`, `__FB_DEBUG__`, etc.):

A callback is used to retrieve their "value". For example: `__LINE__`'s callback simply returns a string containing the lexer's current line number.

Expansion

The macro expansion text (`defText`) is stored by the lexer, and now it will read characters from there for a while, instead of reading from the file

input buffer. Skipping chars in the macro text is like skipping chars in the file input: Once skipped it's lost, there is no going back. So, there never "old" (parsed) macro text, only the current char and to-be-parsed text. New macro text is prepended to the front of existing macro text. That way macros inside macros are expanded.

This implementation does not (easily) allow to detect macro recursion. It would be hard to keep track of which characters in the macro text buffer belong to which macro, but that would be needed to be able to push and pop macros properly. It could be done more easily with a token run implementation as seen in GCC's libcpp. However C doesn't allow recursive macros in the first place: In C, a macro's identifier is undefined (does not trigger expansion) inside that macro's body. That is not the case in fbc, because (again) a way to detect when a macro body ends is not implemented.

Currently fbc only keeps track of the first (oplevel) macro expanded, because it's easy to detect when that specific macro's end is reached: as soon as there is no more macro text.

That's why the recursion is detected here:

```
#define a a
a
```

and here too:

```
#define a b
#define b a
a
```

but not here: (Note that fbc will run an infinite loop)

```
#define a a
#define m a
m
```

Directive parsing



Preprocessor directives (`#if`, `#define`, `#include`, etc.) are parsed during `pp.bas:ppCheck()`. After moving to the next token (or loading a new token current token is a `'#'`. If so it will also check whether the previous token begin, and directly parses the PP directive, using the same `lexGetToken()` parser. This is necessary because some PP directives result in parser function `identifier.bas:cIdentifier()` is used by the `#ifdef` parser, to recognize

```
dim as integer i
#ifdef i
#print yes, the variable will be recognized
#endif
```

So, `lexSkipToken()` is recursive because of the PP. `ppCheck()` will only call `lexSkipToken()`, but not if it was called recursively from the PP. This lets `#macro ... #endmacro` or `skip #if ... #endif` blocks without "executing" the code. Note that unlike C, FB allows macros to contain PP directives.

As a result, every time the FB parser skips an EOL, `lexSkipToken()` might then call the PP to let it parse that directive. It may "silently" parse more directives than the PP directives are even there. The PP parsing launched from `lexSkipToken()` and `#include` and call `fb.bas:fbIncludeFile()` to parse it immediately, recurses to `toplevel.bas:cProgram()` for that `#include` file. The parser has to be able to handle this during every `lexSkipToken()` at EOL, but luckily that is not a big deal. It keeps track of compound statements anyways.

Note that PP directives are not handled during token look ahead (`lex.bas:lexLookAhead()` to look ahead across EOL, it could very well see a PP directive. Luckily that is not necessary.

Macro expansion in PP directives

The beginning of directives, the keyword following the `'#'`, is parsed with `lexSkipToken()`. Redefining PP keywords (intentionally) has no effect on the PP directives

```
#define define foo
#define bar baz
```

will *not* intermediately be seen as:

```
#foo bar baz
```

Directives like `#if` & co. make use of the PP expression parser, which do not do macro expansion at the point of PP expressions. For example:

```
#define foo 1
#if foo = 1
#endif
```

The `#define` and `#macro` directives don't do macro expansion at all. A macro is just a text substitution.

#define/#macro parsing

`pp.bas:ppDefine()` first parses the macro's identifier. If there is a '(' following the identifier, the parameter list is parsed too.

Then the macro body is parsed. For each token, its text representation is appended to the macro body text. Space is preserved (but trimmed); consecutive empty lines are removed.

If the macro has parameters, the macro tokens will be created (as discussed in the next section). Macro parameters are added to a temporary hash table, which associates each parameter with its index. Then, identifiers in the macro body are looked up, and when a parameter identifier is found, a macro token is created, instead of appending the token to the previous text. After that parameter(index), if there is other text again, a new macro token is created.

Using `#` on a parameter results in the creation of a `stringify_parameter(index)` macro token. The `##` operator is simply omitted from the macro body, so `a##b` becomes `a b`. The `##` operator before/after/between parameters goes into `text()` macro tokens.

For example:

```
#define add(x, y) foo bar x + y
```

And the actions of the #define/#macro parser will be:

```
'add'      - The macro's name
'('        following the name, without space in between: Parse the param
'x'        - Parameter 0.
','        - Next parameter.
'y'        - Parameter 1.
')'        - End of parameter list.
Create the macro body in form of macro tokens.
' '        - Create new text(" ").
'foo'      - Append "foo".
' '        - Append " ".
'bar'      - Append "bar".
' '        - Append " ".
'x'        - Is parameter 0, create new param(0).
' '        - Create new text(" ").
'+'        - Append "+".
' '        - Append " ".
'y'        - Is parameter 1, create new param(1).
EOL        - End of macro body.
```

Resulting in this macro body:

```
text(" foo bar "), param(0), text(" + "), param(1)
```

The #define parser allows macros to be redefined, if the body is the same

```
#define a 1
#define a 1
```

does not result in a duplicated definition. However this would:

```
#define a 1
#define a 2
```

Since those are pure text #defines, the comparison of the bodies is as simple as implemented for macros with parameters currently.

PP expressions

The preprocessor has its own (but fairly small and simple) expression parser that works much like `parser-expression.bas:cExpression()`, except instead of just parsing, it immediately evaluates the expressions.

PP skipping

The preprocessor uses a simple stack to manage `#if/#endif` blocks. The `#includes` in them, but they cannot go across files. False blocks (`#if 0`, or `#ifdef` blocks that are not defined) are skipped when parsing the `#if 0` or the `#else` (`pp-cond.bas:ppSkip()`), before the `#endif`.

For example:

```
#if 1          (push to stack: is_true = TRUE, #else not visited)
...          (will be parsed)
#else         1) Set the #else visited flag for the current stack node
              so further #else's are not allowed.
              2) Since the current stack node has is_true = TRUE,
                 that means the #else block must be skipped, -> call ppSkip()
...          (skipped in ppSkip())
#endif       (parsed from ppSkip(), skipping ends, ppSkip() returns to lexSkipToken())
```

Note that there are a few tricky bits about PP skipping. Since macro expansion must be done even during PP skipping, because an `#else` or `#ifdef` block may contain a macro definition. Also, multi-line `#macro` declarations are not handled during PP skipping.

```
#if 0
#macro test()
#endif
#endmacro
```

will be seen as:

```
#if 0
#macro test()
```

```
#endif  
#endmacro
```

Resulting in an error (#endmacro without #macro).

So, this:

```
#if 0  
#macro test()  
    #endif  
#endmacro  
#endif
```

will not work as suggested by the indentation.

File contexts



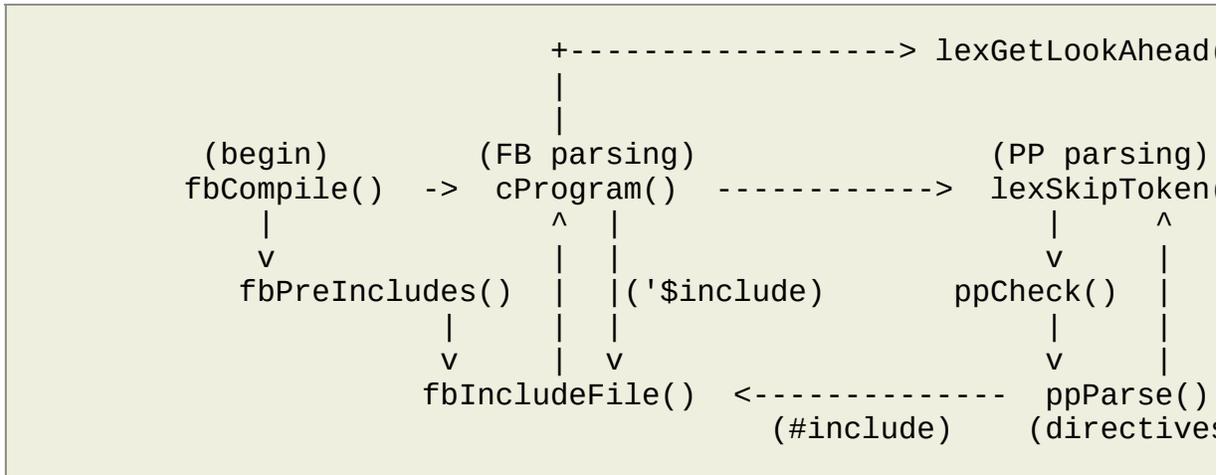
Because `#includes` can occur in the middle of input files, the lexer needs to push file contexts to a stack. File input buffer, macro expansion buffer and the token queue form a so-called "context". It is file specific and thus it must be pushed onto a stack, so that the lexer can return to the parent (after parsing an `#include`), without losing any tokens or macro text. Note that macros can contain `#includes` too.

```
fb.bas:fbIncludeFile() basically just consists of:  
lexPush()  
cProgram()  
lexPop()
```

Quick overview of the call graph



Showing the recursion between the FB parser, the PP parser, and the le



Grammar Notation

Format of a production

left hand side: right hand side;

: should be read as 'is defined as'.

The right hand side of a production is terminated by a ;.

A word in *italics* represent the name of a production (the left hand side of the production).

Few operators are used to describe the FreeBASIC grammar.

operator	meaning
.	any character
*	0 or more (repetition)
+	1 or more (repetition)
?	optional (choice)
()	grouping
	separator (separates alternatives)
semicolon	end of production

Any symbol that appears on the right hand side of a production that is not an operator and does not appear in *italics* represents itself and appears **bold**.

A symbol at the right hand side of a rule can refer to a production. Such references are in *italics*.

For navigational purposes a reference is a link to the production being referenced.

When reading the grammar be aware that FreeBASIC is a case insensit language.

The grammar presented is not an exact statement of the FreeBASIC lan

Go straight to:

program
expression

Tokens

white: **lt** |
any_char: **any valid character**;
eol: **\n|\r|\n\r**;
statement_separator: **(: | *eol*)+**;
dot: **.**;
sign: **+|-**;
alpha: **a|b|c|d|e|f|g|h|i|j|k|l|m|n|p|q|r|s|t|v|w|x|y|z**;
digit: **0|1|2|3|4|5|6|7|8|9**;
hexdigit: **a|b|c|d|e|f|*digit***;
octdigit: **0|1|2|3|4|5|6|7**;
bindigit: **0|1**;
alphadigit: ***alpha*| *digit***;
integer_suffix: **%|&|l|u|ul|ll|ull**;
floating_point_suffix: **!|#|f**;
suffix: ***integer_suffix*|*floating_point_suffix*|\$**;
expchar: **d|e**;
operator
: = | < | > | <> | + | - | * | @
& | -> | / | \ | ^ | andalso
orelse | and | or | xor | eqv | imp
+= | -= | *= | /= | \= | ^= | &= |
and= | or= | xor= | eqv= | imp=
new | delete | delete[] | cast | procptr
varptr | strptr | sizeof | [] | ()
;

binary_operator
: = | < | > | <> | + | - |
& | -> | / | \ | ^
+= | -= | *= | /= | \= | ^= | &= |

and= | or= | xor= | eqv= | imp=
andalso | orelse
;

identifier
: **(alpha)(alphadigit|_)***
nbsp **_(alphadigit|_)+**
;

literal
: **sign integer_literal integer_suffix**
nbsp **sign floating_literal floating_point_suffix**
nbsp **string_literal**
;

integer_literal
: **decimal_integer**
hexadecimal_integer
octal_integer
binary_integer
;

decimal_integer: **digit+**;
hexadecimal integer: **&hhexdigit+**;
octal_integer: **&oocdigit+**;
binary_integer: **&bbindigit+**;

floating_literal
: **digit+(dot(digit+)?)(exp_char?(sign?digit+)?)?suffix?**
nbsp **(dot(digit+)?)(exp_char?(sign?digit+)?)?suffix?**
;

string_literal
: **(!|\$)?" (escape_sequence|""|any_char)*" (white*string_literal)***
;

escape_sequence
: **simple_escape_sequence**
unicode_escape_sequence

decimal_escape_sequence
hexadecimal_escape_sequence
octal_escape_sequence
binary_escape_sequence
;

simple_escape_sequence
: **\a\b\f|\l|\n|\r|\t|\v|\|\'|\"**
;

unicode_escape_sequence
: **\uhexdigit hexdigit hexdigit hexdigit**
;

decimal_escape_sequence
: **\ldigit digit digit**
;

hexadecimal_escape_sequence
: **\&h;hexdigit hexdigit**
;

octal_escape_sequence
: **\&o;octdigit octdigit octdigit**
;

binary_escape_sequence
: **\&b;bindigit bindigit bindigit bindigit bindigit bindigit bindigit bind**
;

Comment

comment
: (' | **rem**) ((\$*directive*) | (**any_char_but_eol***))
;
multiline_nested_comment
: ! (. | **multiline_nested_comment**)* !;

Toplevel

program

: *line** EOF?

;

line

: *label* (*statement*|*namespace_statement*)? *comment*? *eol*

;

label

: *identifier* :

;

statement

: *statement_separator*?

(*declaration* | *procedure_call_or_assign* | *compound_statement* | *qi assignment*)?

(*statement_separator statement*)*

;

declaration

:(*public*|*private*)?

(

(*static*

(*function_definition*

sub_definition

operator_definition

constructor_definition

destructor_definition

property_definition

variable_declaration

)

)

function_definition

sub_definition

destructor_definition

property_definition

constructor_definition

operator_definition

const_declaration
type_or_union_declaration
variable_declaration
enumeration_declaration
auto_variable_declaration
)
declare *procedure_declaration*
;

procedure_call_or_assign
: **call** *identifier* ((*procedure_parameter_list*)?)
identifier *procedure_parameter_list*?
(*identifier* | **function** | **operator** | **property**) = *expression*
;

compound_statement
: *namespace_statement*
scope_statement
if_statement
for_statement
do_statement
while_statement
select_statement
;

namespace_statement
: **namespace** *identifier* (**alias** *string_literal*)? (*declaration* | *namespa*
;

scope_statement: **scope** *statement_separator* *statement** **end scope**
;

if_statement
: *short_if_statement* | *long_if_statement*
;

short_if_statement
: **if** *expression* **then** *statement_separator* *statement*

else *statement_separator statement**
(*eol*| end if | endif)
;

long_if_statement
: **if** *expression* **then** *statement_separator*
*statement**
*elseif_block**
(**else** *statement_separator statement**)?
(end if|endif)
;

elseif_block
: **elseif** *expression* **then** *statement_separator statement**
;

for_statement
: **for** *identifier* (as *scalar*)? = *expression* to *expression* (step *expression*:
(*statement*|exit for(, for)* | continue for (, for)*)* next *identifier* (, *identifier*)
;

do_statement
: **do** (until|while) *expression* (*statement*|exit do (, do)* | continue do
do (*statement*|exit do (, do)* | continue do (, do)*)* loop (until|while)
;

while_statement
: **while** *expression* *statement_separator*
(*statement* | exit while (, while)* | continue while (, while)*)*
wend
;

select_statement
: **select case** (as const) *expression* *case_statement** **case else** *statement*
;

case_statement
: **case** *case_expression* (, *case_expression*)*
;

case_expression

: *expression* | *expression to expression* | is (> | < | >= | <= | = | <>) e)

assembler_block

: **asm** *comment?* (*asm_code* *comment?* *eol*)+ **end asm**
;

assignment

: **let?** *variable* *binary_operator* = *expression*
variable (*procedure_parameter_list*)
;

variable

: *highest_precedence_expression*;

const_declaration

: **const** (**as** *symbol_type*)? *const_assign* (, *const_assign*)*
;

type_or_union_declaration

: *type_declaration* | *union_declaration*
;

type_declaration

: **type** *identifier* (**alias** *string_literal*)? (**field = expression**)? (**comment**
type_member_declaration+
end type)
;

union_declaration

: **union** *identifier* (**alias** *string_literal*)? (**field = expression**)? (**comment**
union_member_declaration+
end union)
;

type_member_declaration

: ((**union|type**) *comment?* *statement_separator* *element_declaration*
end (**union|type**)
)
element_declaration

as *as_element_declaration*
;

variable_declaration
: (redim preserve?*|dim|common*) shared? *symbol_type*
extern import? *symbol_type* alias *string_literal*
static *symbol_type*
;

symbol_type
: const? unsigned?
(
scalar
string (* *integer_literal*)?
wstring (* *integer_literal*)?
user_defined_type
function ((*parameters*)) (as *symbol_type*)
sub ((*parameters*))
) (const? (ptr|pointer))*
;

scalar
: byte
ubyte
short
ushort
integer
uinteger
longint
ulongint
long
ulong
single
double
;

parameters
: *parameter* (, *parameter*)*

;

parameter

: **(byval|byref)? (identifier (())?)? as symbol_type (= literal)?**

;

user_defined_type

: **identifier**

;

procedure_declaration

: **static?**

(sub_declaration|function_declaration|constructor_declaration|des

;

procedure_parameter_list

: **procedure_parameter (, procedure_parameter)***

;

procedure_parameter

: **byval? (identifier(())? | expression)**

;

expressions

expression

: **boolean_expression**

;

boolean_expression

: **logical_expression((andalso | orelse) logical_expression)***

;

logical_expression

: **logical_or_expression** ((**xor** | **eqv** | **imp**) **logical_or_expression**)^{*}
;

logical_or_expression
: **logical_and_expression**(**or** **logical_and_expression**)^{*}
;

logical_and_expression
: **relational_expression** (**and** **relational_expression**)^{*}
;

relational_expression
: **concatenation_expression** ((=**>**|**<**|**<>**|**<=**|**>=**) **concatenation_expre:**
;

concatenation_expression
: **add_expression**(**&** **add_expression**)^{*}
;

add_expression
:
: **shift_expression**((**+** | **-**) **shift_expression**)^{*}
;

shift_expression
:
: **mod_expression** ((**shl** | **shr**) **mod_expression**)^{*}
;

mod_expression
: =
: **integer_division_expression**(**mod** **integer_division_expression**)^{*}
;

integer_division_expression
: **multiplication_expression** (\ **multiplication_expression**)^{*}
;

multiplication_expression
: **exponentiation_expression** (**(* | /)** **exponentiation_expression**)
*
;

exponentiation_expression
: **prefix_expression** (**^** **prefix_expression**)
*
;

prefix_expression
: **(-|+)** **exponentiation_expression**
not relational_expression
highest_precedence_expression
;

highest_precedence_expression
: **address_of_expression**
(dereference_expression | casting_expression |
pointer_type_casting_expression | parenthesised_expression)
anonymous_udt
atom
;

address_of_expression
: **varptr** (**highest_precedence_expression**)
procptr (**identifier** (**()**) ?)
@ (**identifier** (**()**) ? | **highest_precedence_expression**)
sadd|strptr (**expression**)
;

dereference_expression
: ***+** **highest_precedence_expression**
;

casting_expression
: **cast** (**symbol_type** , **expression**)
;

quirk_function

: quirk_function_name procedure_parameter_list
;

quirk_function_name

**: mkd | mki | mkl | mklongint | mkshort
cvd | cvi | cvl | cvlongint | cvs | cvshort
asc | chr | instr | instrev | lcase | left | len | lset | ltrim | mid | right |
rset | rtrim | space | string | ucase | wchr | wstr | wstring
abs | sgn | fix | frac | len | sizeof, sin | asin | cos | acos | tan | atn | sqr
| log | exp | atan2 | int
peek
lbound | ubound
seek | input | open | close | get | put | name
err
iif
va_first
cbyte | cshort | cint | clng | clngint | cubyte | cushort | cuint | culng |
culngint | csng | cdbl | csign | cunsg
type
view | width | color | screen
;**

quirk_statement

: jump_statement

print_statement

data_statement

array_statement

line_input_statement

input_statement

poke_statement

file_statement

write_statement

error_statement

on_statement

view_statement

mid_statement

lrset_statement

width_statement
color_statement
gfx_statement

;

jump_statement
: **goto** *identifier*

;

print_statement
: (**print** | **?**) (**#** *expression* ,)? (**using** *expression* ;)? (*expression*? ; | ,)*;

data_statement
: **restore** *identifier*
read *variable* (, *variable*)*
data *literal* (, *literal*)*

;

array_statement
: **erase** *variable* (, *variable*)*
swap *variable* , *variable*

;

line_input_statement
: **line input** ;? (**#** *expression* | *expression*?) (, | ;)? *variable*?

;

input_statement
: **input** ;? ((**#** *expression* | *string_literal*) (, | ;))? *variable* (, *variable*)*

;

poke_statement
: **poke** *expression* , *expression*

;

file_statement
: **close** (**#?** *expression*) (, **#?** *expression*)*
seek **#?** *expression* , *expression*

put # expression , expression? , expression
get # expression , expression? , variable
(lock|unlock) #? expression , expression (to expression)?
name expression as expression
;

write_statement
: **write (# expression)? (expression? ,)***
;

error_statement
: **error expression**
err = expression
;

on_statement
: **on local? (error | expression) goto identifier**
;

view_statement
: **view (print (expression to expression)?)**
;

mid_statement
: **mid (expression , expression (, expression) = expression**
;

lreset_statement
: **lset|rset highest_precedence_expression ,**
highest_precedence_expression
;

width_statement
: **width expression , expression**
width lprint expression
width (# expression| expression), expression
;

color_statement

: color *expression* , *expression*
;

gfx_statement

: pset (*expression* ,)? step? (*expression* , *expression*) (, *expression*)?

line (*expression* ,)? step? ((*expression* , *expression*))? - step? (*expression* , *expression*) (, *expression*? (, *string_literal*? (, *expression*)?)?)?

circle (*expression* ,)? step? (*expression* , *expression*) , *expression* ((, *expression*? (, *expression*? (, *expression*? (, *expression* (, *expression*)?)?)?)?)?)?

paint (*expression* ,)? step? (*expression* , *expression*) (, *expression*? (, *expression*?))

draw (*expression* ,)? *expression*

view (screen? (*expression* , *expression*) - (*expression* , *expression*) (, *expression*? (, *expression*)?)?)?

palette get? ((using *variable*) | (*expression* , *expression* (, *expression* , *expression*)?)?)

put (*expression* ,)? step? (*expression* , *expression*) , ((*expression* , *expression*) - (*expression* , *expression*) ,)? *variable* (, *expression* (, *expression*)?)?

get (*expression* ,)? step? (*expression* , *expression*) - step? (*expression* , *expression*) , *variable*

screen (*integer_literal* | ((*expression* (((, *expression*)? , *expression*)? *expression*)? , *expression*))

screenres *expression* , *expression* (((, *expression*)? , *expression*)? , *expression*)?

;