

Inno Setup Preprocessor: Introduction

Inno Setup Preprocessor (ISPP) is a preprocessor add-on for Inno Setup.

The main purpose of ISPP is to automate compile-time tasks and decrease the probability of typos in your scripts. For example, you can declare an ISPP variable (compile-time variable) – your application name, for instance – and then use its value in several places of your script. If for some reason you need to change the name of your application, you'll have to change it only once in your script. Without ISPP, you would probably need to change all occurrences of your application name throughout the script (AppName, AppVerName, DefaultGroupName etc. [\[Setup\] section](#) directives).

Another example of using ISPP would be gathering version information from your application by reading the version info of an EXE file, and using it in AppVerName [\[Setup\] section](#) directive or anywhere else. Without ISPP, you would have to modify your script each time version of your application changes.

Also, conditional in- and exclusion of portions of script is made possible by ISPP: you can create one single script for different versions/levels of your applications (for example, trial versus fully functional).

Finally, ISPP makes it possible to split long lines using a line spanning symbol.

Note: ISPP works exclusively at compile-time, and has no run-time functionality.

All topics

- [Documentation Conventions](#)
- [Directives](#)
- [Functions](#)
- [Predefined Variables](#)
- [Line Spanning](#)

- [Example Script](#)
- [User Defined Macros](#)
- [ISPPBuiltins.iss](#)
- [Visibility of Identifiers](#)
- [Expression Syntax](#)
- [Extended Command Line Compiler](#)
- [Translation](#)
- [Current translation](#)

Inno Setup Preprocessor: Documentation Conventions

Directive syntax documenting conventions

Directive usage syntax uses the following conventions.

()	Group of tokens.
[]	Optional token or group of tokens.
	Mutually exclusive tokens.
. . .	Previous token or group of tokens can be repeated.
token	Reserved word or symbol(s). Must be typed exactly as shown.
<token>	Non-terminal. Its syntax is either shown before, or explained.

Function prototypes documenting conventions

Function prototypes are shown as function result type, function name and list of formal arguments in parentheses.

Words `int`, `str`, `any`, and `void` are used to specify integer type, string type, any type, or null type (also referred to as nothing, `void`), respectively. Null type as function result means that function does not return any value.

Question mark (?) after the type of an argument means that this argument is optional.

Inno Setup Preprocessor: Directives

In ISPP directives can be used in two ways: simple or inline.

Simple directives occupy a whole line and begin with the # symbol. For example the following defines a variable called MyAppName:

```
#define MyAppName "My Program"
```

Inline directives appear inside other lines and begin with {# and end with }. For example the following sets Inno Setup's AppName directive to the value of the previously defined variable:

```
[Setup]  
AppName={#MyAppName}
```

As seen in the above example it is not necessary to specify the name of the [emit](#) directive when it is used inline, so {#MyAppName} is short for {#emit MyAppName}.

Available directives

- [#define](#)
- [#dim](#)
- [#undef](#)
- [#include](#)
- [#file](#)
- [#emit](#)
- [#expr](#)
- [#insert](#)
- [#append](#)
- [#if, #elif, #else, #endif](#)
- [#ifdef, #ifndef, #ifexist, #ifnexist](#)
- [#for](#)

- #sub, #endsub
- #pragma
- #error

Inno Setup Preprocessor: Functions

There are a number of predefined functions provided by ISPP which you can use to perform compile-time actions and/or change your script. For example the following reads version info from an EXE and uses the return value of the function to change the script:

```
#define MyAppVer  
GetFileVersion(AddBackslash(SourcePath) +  
"MyProg.exe")
```

```
[Setup]  
AppVerName=MyProg version {#MyAppVer}
```

Available functions

- [GetFileVersion](#)
- [GetStringFileInfo](#)
- [Int](#)
- [Str](#)
- [FileExists](#)
- [FileSize](#)
- [ReadIni](#)
- [WriteIni](#)
- [ReadReg](#)
- [Exec](#)
- [Copy](#)
- [Pos](#)
- [RPos](#)
- [Len](#)
- [SaveToFile](#)

- [Find](#)
- [SetupSetting](#)
- [SetSetupSetting](#)
- [LowerCase](#)
- [EntryCount](#)
- [GetEnv](#)
- [DeleteFile](#)
- [CopyFile](#)
- [FindFirst](#)
- [FindNext](#)
- [FindClose](#)
- [FindGetFileName](#)
- [FileOpen](#)
- [FileRead](#)
- [FileReset](#)
- [FileEof](#)
- [FileClose](#)
- [GetDateTimeString](#)
- [GetFileDateTimeString](#)
- [GetMD5OfString](#)
- [GetMD5OfUnicodeString](#)
- [GetMD5OfFile](#)
- [GetSHA1OfString](#)
- [GetSHA1OfUnicodeString](#)
- [GetSHA1OfFile](#)

- [Trim](#)
- [StringChange](#)
- [Defined](#)
- [TypeOf](#)

Inno Setup Preprocessor: Predefined Variables

There are a number of predefined variables provided ISPP:

<code>__COUNTER__</code>	int . Automatically increments each time it is used (afterwards).
<code>__FILE__</code>	str . Returns the name of the current file. Empty string for the root file.
<code>__INCLUDE__</code>	str . Returns the current include path (or paths delimited with semicolons) set via <code>#pragma include</code> .
<code>__LINE__</code>	int . Returns the number of the line in the current file (not a translation) on which the variable is used (or macro that uses this variable is called).
<code>__OPT_X__</code>	void . Defined if specified option set via <code>#pragma option -x+</code> is in effect. In place of "X" may be any letter from "A" to "Z." Use Defined function to test whether the variable is defined.
<code>__PATHFILENAME__</code>	str . Similar to <code>__FILE__</code> but, returns the full pathname of the file. Empty string for the root file.
<code>__POPT_X__</code>	void . Defined if specified parser option set via <code>#pragma parseroption -x+</code> is in effect. In place of "X" may be any letter from "A" to "Z." Use Defined function to test whether the variable is defined.
<code>__WIN32__</code>	void . Always defined.
<code>ISPP_INVOKED</code>	void . Always defined.
<code>PREPROCVER</code>	int . Returns the 32-bit encoded version of ISPP. Highest byte holds the major version, lowest byte holds the build number.
<code>WINDOWS</code>	void . Always defined.
<code>UNICODE</code>	void . Always defined for Unicode ISPP. Use Defined function to test whether the variable is defined.
<code>CompilerPath</code>	str . Points to the directory where the

SourcePath	compiler is located. str . Points to the directory where the current script is located, or the My Documents directory if the script has not yet been saved.
Ver	int . Returns the 32-bit encoded version of Inno Setup compiler. Highest byte holds the major version, lowest byte usually holds zero.

Inno Setup Preprocessor: Line Spanning

By ending lines with ISPP's line spanning symbol preceded with a space, you can split long lines. For example:

```
#define MyAppName \  
    "My Program"
```

The default line spanning symbol is "\" which can be changed using [pragma](#).

Inno Setup Preprocessor: Example Script

An example script called *ISPPExample1.iss* is located in a separate folder. Please click the "Inno Setup Example Scripts" shortcut created in the Start Menu when you installed Inno Setup, or open the "Examples" folder in your Inno Setup directory.

Also see [ISPPBuiltins.iss file](#).

Inno Setup Preprocessor: User Defined Macros

You can define so called "macros" in your scripts, these can be thought of as "user defined functions."

Macro declaration consists of formal parameter list and expression. That expression is evaluated when macro is called (see below). The result of the macro call is the result of the macro expression. Macro expression can contain parameter names, they are treated as usual variables.

The formal syntax of macro declaration is provided in [define](#) and the [ISPPBuiltins.iss file](#) contains many example macros.

Please note that there must be no space between macro name and opening parenthesis.

Actual parameters for parameters declared as by-reference must be modifiable l-values (in other words, other defined variables or expressions that evaluate to l-values). If macro expression modifies by-reference parameter, the variable that is passed as this parameter gets modified. By-value parameters can also be modified by macro expression (using assignment operators), but this modification doesn't affect the value of a variable which could be passed as this parameter.

Though macro can only contain one expression, it can be used as full featured user defined function, because ISPP supports sequential evaluation operator (comma), assignment operators (simple and compound) and conditional operator (? :).

See also

- [Macros' Local array](#)

Inno Setup Preprocessor: ISPPBuiltins.iss

The ISPPBuiltins.iss file is accompanying the Inno Setup Preprocessor. It is automatically included, if it exists in the compiler directory, as if the very first line of your script contained an `include` directive for it. This file contains common declarations, such as special constants for using with functions, and some useful macros. The file is a regular Inno Setup Script file but mostly contains only ISPP directives.

To learn more about the functionality provided by this file please open it with the Inno Setup Compiler, it is well commented.

Inno Setup Preprocessor: Visibility of Identifiers

Variables (as well as macros, read "variable or macro" anywhere it says "variable" below) can be explicitly defined as "public," "protected," or "private." To define such a variable, its name in its `define` directive should be prepended with one of the visibility keywords:

```
#define public MyVar 12
#define protected MyVar 13
#define private MyVar 14
```

In the example above, none of the last two declarations undefine any of the previous, though they share the same identifier (MyVar). This is because they are declared in different visibilities.

Public variables are ordinary variables accessible from anywhere after the point they are declared.

Protected variables are accessible only in the file they are declared in and in files included by that file via `include` or `file` directives. You can basically think of them as public variables which are automatically undefined once their file has finished.

Private variables are accessible only in the file they are declared in. They are not propagated to any other file, be it included or "parent" file.

Since ISPP does not have semantics of pushing and popping variable value, visibility resolution can be useful.

Note that you cannot explicitly refer to a variable in a specific visibility from expressions. Given the example above, if MyVar is mentioned in expression in declaration file, its identifier refers to private MyVar. If it is mentioned in included file, it refers to protected MyVar. If it is mentioned in one of the files above the declaration file on the include stack (i. e. one of the files from which a chain of `include` directives resulted in processing the declaration file), it refers to public MyVar.

Also note, that if we'd swap last two declarations from the above example, private MyVar would become inaccessible (until protected is undefined) because protected would be declared after it and would take

precedence. But it wouldn't undefine its private counterpart.

Each file can set a default visibility, the visibility that will be used when no resolution clause is specified in variable declaration. This can be done using `define` directive, for example:

```
#define protected
```

sets protected visibility by default.

The default visibility isn't used when evaluating expressions, it is only used when a variable is defined or undefined without explicitly specifying its visibility. When default visibility is not set, public is assumed by default. Setting default visibility is not propagated on included or parent files.

In macro expressions, avoid using identifiers of lower visibility than the one macro is declared in. This may cause "Undeclared identifier" errors if macro is called from another file.

It is recommended that you use appropriate visibility when declaring variables to avoid problems with unexpected redefinition of a variable (for example in included third-party file). If no included files depend on a variable, declare it as private. If they do, but the parent file doesn't, declare it as protected. Declare it as public otherwise. If you're unsure, then protected visibility is the common case.

Inno Setup Preprocessor: Expression Syntax

ISPP uses C/C++-like expression syntax. It supports simple and compound assignment operators, conditional operator, and sequential evaluation operator. Although ISPP is an interpreter, it does support short circuit boolean evaluation and never evaluates expressions (nor calls any macros mentioned in those expressions) that should not be evaluated due to specific rules (for example, when conditional operator is used, always only 2 out of 3 operands are evaluated).

The [ISPPBuiltins.iss file](#) contains many example expressions.

Differences between C and ISPP expression syntax

- ISPP does not support a number of operators (reference, dereference, namespace resolution, member selection, etc.).
- ISPP treats an identifier and the equality sign as a name of an argument, if it is used in argument list.
- Arithmetic division operator (slash) performs integer division, since ISPP does not support floating point math.
- ISPP does not check for validity of expressions in certain cases. For example, in conditional expression, "true" operand can be of string type, whereas "false" operand can be of integer type.
- String literals can be quoted by both single and double quotes (in both modes – C-style or Pascal-style). If a literal begins with a single quote, it must also end with a single quote. Double quotes may be used in single quoted string without any escaping, and vice versa. Within a string the character used to quote the string must be escaped (the manner depends on current state of "Pascal-style string literals" parser option, see [pragma](#)).

Data types

There are three types in ISPP: void, integer, and string. Variable of void type is declared by just specifying its name after `define` directive without any value. Such variables should be used with `ifdef` directive or `Defined` function.

If "allow undeclared identifiers" parser option is off (the default state, see `pragma`), an error is raised when undefined variable is mentioned. Otherwise, it will be treated as a value of type void.

Void is compatible with integer and string in expressions. For example, you can use addition operator with void and integer operands, in this case void operand will be treated as zero. In conjunction with string, void operand is treated as an empty string.

Comments

Comments may be embedded in expression by using a slash and an asterisk. For example:

```
#emit Var1 /* this is a comment */ + Var2 /* this is  
a comment */
```

Also one line comments are supported. Those comments must begin with a semicolon. Whole text after the semicolon up to the end of a line is considered comment.

```
#emit Var1 + Var2 ; this is a comment
```

Please note that line spanning feature is triggered before any further processing, thus a comment may occupy more than one line:

```
#emit Var1 + Var2 ; this is \  
a comment
```

Extended Macro Call Syntax

In ISPP it is possible to use named arguments when calling user defined macro. Given the declaration:

```
#define MyMacro(int A = 2, int B = 2) A + B
```

This macro can be called specifying argument names:

```
#emit MyMacro(A = 5, B = 10)
```

```
#emit MyMacro(B = 3)
```

```
#emit MyMacro(B = 10, A = 5)
```

- If a name is specified for one argument, then all (required) arguments in the list must also be named.
- The order of named arguments does not matter.
- Because of this extension, an assignment expression must be enclosed in parentheses, if not using extended call syntax, to avoid ambiguity:

```
#emit MyMacro((MyVar = 5), 10)
```

In the above example, the equality sign is treated as a direct assignment operator.

Although functions do not have named arguments, it is still required to enclose assignment expressions in parentheses when calling those functions.

- By standard rule comma is used to separate actual parameters. If you need to use sequential evaluation operator, you must include the expression in parentheses:

```
#emit MyMacro((SaveToFile("script.txt"), 5), 10)
```

In the above example, the first comma is treated as the sequential evaluation operator, whereas the second one as the argument delimiter.

Inno Setup Preprocessor: Extended Command Line Compiler

Inno Setup Preprocessor replaces the standard Inno Setup Command Line Compiler (ISCC.exe) by an extended version. This extended version provides extra parameters to control Inno Setup Preprocessor.

Usage: `iscc [options] scriptfile.iss`. Or to read from standard input: `iscc [options] -`.

Options are to emulate a [define](#) or [pragma](#) directive are:

<code>/d<name>[=<value>]</code>	Sets #define public <name> <value>
<code>/\$<letter>(+ -)</code>	Sets #pragma option -<letter> (+ -)
<code>/p<letter>(+ -)</code>	Sets #pragma parseroption - <letter>(+ -)
<code>/i<paths></code>	Sets #pragma include -<paths>
<code>{#<string></code>	Sets #pragma inlinestart - <string>
<code></code> }<string>	Sets #pragma inlineend -<string>
<code>/v<number></code>	Sets #pragma verboselevel - <number>

Other valid options are: `"/O` to specify an output path (overriding any `OutputDir` setting in the script), `"/F` to specify an output filename (overriding any `OutputBaseFilename` setting in the script), and `"/Q` for quiet compile (print only error messages).

Example: `iscc /$c- /pu+ "/dLic=Trial Lic.txt"
/iC:\INC;D:\INC "c:\isetaup\samples\my script.iss"`

ISCC will return an exit code of 0 if the compile was successful, 1 if the command line parameters were invalid or an internal error occurred, or 2 if the compile failed.

Inno Setup Preprocessor: Translation

Translation refers to the preprocessed script.

Inno Setup Preprocessor: Current translation

Current [translation](#) refers to current output of ISPP, the translated (preprocessed) part of the script up to the point (or line) which ISPP is currently processing.

Inno Setup Preprocessor: #emit

Syntax

emit-directive: **(emit | =)** <expr>

Description

Replaces the directive with the value of `expr`.

When used inline, the name of this directive can be omitted unless `expr` begins with the name of another directive.

Examples

[Files]

```
#emit 'Filename: "file1.ext"; DestDir: {' +  
MyDestDir + '}'
```

```
Filename: "file2.ext"; DestDir: {{#MyDestDir}}
```

```
#emit GenerateVisualCppFilesEntries ; user defined  
macro
```

[Code]

```
const
```

```
  AppName = '{#SetupSetting\("AppName"\)}';
```

See also

[expr.](#)

Inno Setup Preprocessor: #define

Syntax

define-directive: <variable-definition>
 <macro-definition>
 <default-visibility-set>

variable-definition: (**define** | :) [**private** | **protected** | **public**]
 <ident> [[<expr>]] [[=] <expr>]

macro-definition: (**define** | :) [**private** | **protected** | **public**] <ident> (
 [<formal-macro-args>]) <expr>

default-visibility-set: (**define** | :) **private** | **protected** | **public**

formal-macro-args: <formal-macro-arg> [, <formal-macro-arg>]...

formal-macro-arg: <by-val-arg> | <by-ref-arg>

by-val-arg: [<type-id>] <ident> [= <expr>]

by-ref-arg: [<type-id>] * <ident>

type-id: **any** | **int** | **str** | **func**

Description

The first syntax ("variable-definition") defines a variable named `ident`, or assigns a value to an element of an array named `ident`. If none of the `public`, `protected`, or `private` keywords are specified, default [visibility](#) is assumed.

The second syntax ("macro-definition") defines a macro named `ident`. When defining a macro there must be no whitespace between macro name and opening parenthesis, otherwise it will be treated as variable declaration.

The third syntax ("default-visibility-set") sets the default [visibility](#) of further variable and macro definitions in this file. If no visibility declaration occurs in a file, public visibility is assumed by default.

Examples

```
#define MyAppName "My Program"  
#define MyAppVer GetFileVersion("MyProg.exe")  
#define MyArray[0] 15  
#define Multiply(int A, int B = 10) A * B
```

See also

[d_im](#), [undef](#), [Visibility of Identifiers](#).

Inno Setup Preprocessor: #dim

Syntax

dim-directive: **dim** [**private** | **protected** | **public**] <ident> [<expr>]

Description

Declares an array variable and sets its dimension. All elements of the array are initialized to null (void). To assign an element value after declaring the array, use [define](#). Instead of assigning element values with [define](#), it is also possible to set an element value by using it as the left operand of an assignment.

Examples

```
#dim MyArray[10]  
#define MyArray[0] 15
```


See also

[define](#), [undef](#), [Visibility of Identifiers](#).

Inno Setup Preprocessor: #undef

Syntax

undef-directive: **(undef | x) [private | protected | public] <ident>**

Description

Undefines (removes) a variable or macro. If no [visibility](#) (`public`, `protected`, or `private`) is specified, ISPP first tries to remove a private variable of the given name, then protected, then public.

Examples

```
#undef MyVar  
#undef MyMacro  
#undef public MyVar
```

See also

[define](#), [dim](#), [Visibility of Identifiers](#).

Inno Setup Preprocessor: #include

Description

Includes the [translation](#) of the specified file.

If the filename is enclosed in angle brackets, ISPP first searches for the file in the directory where current file resides, then in the directory where the file that included current file resides, and so on. If the file is not found, it is searched on current include path, set via [pragma](#), then on the path specified by INCLUDE environment variable.

If filename is an expression or specified in quotes, it is searched on current include path only.

The filename may be prefixed by "compiler:", in which case it looks for the file in the Compiler directory.

This directive cannot be used inline.

Examples

```
#include <file.iss>  
#include "c:\dir\file.iss"  
#include AddBackslash(CompilerPath) + "common.iss"
```

See also

[file](#), [sub](#).

Inno Setup Preprocessor: #file

Syntax

file-directive: **file** <expr>

Description

Replaces the directive with the name of a temporary file containing the [translation](#) of the specified file. Upon end of compilation, the temporary file is automatically deleted.

Including a file using this directive creates a new independent instance of the preprocessor, passing it options currently in effect and all declared identifiers. If the included file modifies options in some way, they are not propagated back.

When using this directive in Inno Setup's Source parameter of the [Files] section, specify a DestName parameter too, else the file will not be installed with the original name.

This directive can only be used inline.

Examples

```
[Setup]  
LicenseFile={#file "mylic.txt"}
```

See also

[include.](#)

Inno Setup Preprocessor: #expr

Syntax

expr-directive: (**expr** | **!**) <expr>

Description

Evaluates an expression ignoring its result. This directive acts like [emit](#) with the exception that it doesn't emit anything to the [translation](#).

This directive is intended to be used with functions that produce side effects and do not return any significant value.

Examples

```
#expr SaveToFile("preprocessed.iss"),  
Exec("notepad.exe", "preprocessed.iss")
```

See also

[emit.](#)

Inno Setup Preprocessor: #insert

Syntax

insert-directive: **insert** <expr>

Description

Changes the insertion point. By default, each processed line is added to the end of the [translation](#). Using `insert` the point at which the next processed line will be added to the [translation](#) can be changed. `insert` takes an expression which must evaluate to an integer. The insertion point will be set to this integer.

The insertion point is also always automatically incremented each time after line has been added to the [translation](#), so that each new line is inserted after the one previously inserted.

It is not recommended to use script generating functions (such as [SetSetupSetting](#)) which may insert a line by themselves, thus shifting a part of the translation one line down, whereas insertion point is not updated. This may result in different insertion point than expected.

The [Find](#) function can be used to produce values for the `insert` directive.

Examples

```
#insert FindSectionEnd("Icons")  
#insert FindSection("Setup") + 1  
#insert Find(0, "somefile.ext", FIND_CONTAINS)
```

See also

[append.](#)

Inno Setup Preprocessor: #append

Syntax

append-directive: **append**

Description

Resets the insertion point (if previously changed using [insert](#)) to the end of the [translation](#).

See also

[insert.](#)

Inno Setup Preprocessor: #if, #elif, #else, #endif

Syntax

if-directive: **if** <expr>

elif-directive: **elif** <expr>

else-directive: **else**

endif-directive: **endif**

Description

The `if`, `elif`, `else`, and `endif` conditional directives control in- and exclusion of portions of script.

ISPP first evaluates the expressions following each `if` or `elif` directive until it finds one evaluating to non-zero. It then selects the portion of script following this directive up to its associated `elif`, `else`, or `endif`. Earlier portions which followed an `if` or `elif` which evaluated to zero, or which follows any next `elif` are not selected and thus not seen by the Inno Setup compiler.

If no expression evaluated to non-zero, the preprocessor selects the script portion after the `else` directive if present, else nothing is selected.

Finally, after selecting any script portion, ISPP preprocesses it too, so if it contains other preprocessor directives, ISPP carries out those directives as well.

Each `if` directive in a source file must be matched by a closing `endif` directive. Any number of `elif` directives can appear between the `if` and `endif` directives, but at most one `else` directive is allowed. The `else` directive, if present, must be the last directive before `endif`.

The `if`, `elif`, `else`, and `endif` directives can be nested. Each nested `else`, `elif`, or `endif` directive belongs to the closest preceding `if` directive.

Inline conditional directives may not be mixed with simple. If the `if` directive is simple (occupying a whole line), its associated directives (`elif`, `else`, or `endif`) must also be simple and not inline (appearing inside other lines).

Examples

```
#define Lang
```

```
[Tasks]
```

```
#if "English" == Lang =
```

```
ReadIni(SetupSetting("MessagesFile"), \  
  "LangOptions", "LanguageName")
```

```
  Description: "For all users"; Name: all
```

```
#elif "German" == Lang
```

```
  Description: "Fur alle"; Name: all
```

```
#else
```

```
# error Unsupported language
```

```
#endif
```

Inno Setup Preprocessor: #ifdef, #ifndef, #ifexist, #ifnexist

Syntax

ifdef-directive: **ifdef** <ident>

ifndef-directive: **ifndef** <ident>

ifexist-directive: **ifexist** <expr>

ifnexist-directive: **ifnexist** <expr>

Description

You can use the `ifdef`, `ifndef`, `ifexist`, and `ifnexist` directives anywhere `if` can be used. The `ifdef identifier` statement is equivalent to `if 1` when the specified identifier has been defined, and equivalent to `if 0` when the identifier has not been defined or has been undefined with the `undef` directive. These directives check only for the presence or absence of identifiers defined with `define`.

`ifexist` and `ifnexist` directives check for presence and absence of the file, respectively.

Examples

```
[Files]
#ifexist "myfile.ext"
  Filename: "myfile.ext"; DestDir: {app}
#endif
#ifdef Enterprise
  Filename: "extra.dll"; DestDir: {app}
#endif
```

See also

[FileExists](#)

Inno Setup Preprocessor: #for

Syntax

for-directive: **for** { <expr1> ; <expr2> ; <expr3> } <expr4>

Description

Use the for directive to get loop behaviour. for takes 4 expressions. The first expression (expr1) is called "initialization expression," the second expression (expr2) "condition," the third expression (expr3) "action," and the final expression (expr4) "loop body."

The logic the for directive follows is:

1. The initialization expression is evaluated.
2. The condition is evaluated. If it evaluates to 0, the loop ends.
3. The loop body is evaluated.
4. The action is evaluated.
5. Process repeats from 2.

Examples

```
// Call AddFile user defined procedure 200 times  
#for {i = 200; i > 0; i--} AddFile
```

More examples

[FindFirst](#), [FileRead](#).

See also

[sub](#), [include](#).

Inno Setup Preprocessor: #sub, #endsub

Syntax

sub-directive: **sub** <ident>

endsub-directive: **endsub**

Description

sub and endsub directives are used to declare a user defined procedure which is a portion of script which may be included later once or several times. You may think of a user defined procedure as being similar to an external file, and a call to a user defined procedure as being similar to inclusion of an external file, except that procedures may also be called from within expressions. Please note that it is strongly not recommended to call procedures which emit several lines to [translation](#) from within compound expressions or directives.

A procedure is called by simply specifying its identifier, with which it was declared.

A procedure is not processed in any way until it is called, so if any errors exist in its body, they will only pop up when the procedure is called.

Examples

```
#sub AddFile
  #if Copy(FileName, 1, 1) == "A"
    Source: {#FileName}; DestDir: {app}\A
  #else
    Source: {#FileName}; DestDir: {app}
  #endif
#endsub
```

More examples

[FindFirst](#), [FileRead](#).

See also

[User Defined Macros](#), [if](#), [emit](#).

Inno Setup Preprocessor: #pragma

Syntax

- pragma-directive*: <pragma-option>
 <pragma-itokens>
 <pragma-msg>
 <pragma-verblev>
 <pragma-include>
 <pragma-spansymb>
- pragma-option*: **pragma (option | parseroption) -** <letter> (+ | -) [-
 <letter> (+ | -)]...
- pragma-itokens*: **pragma (inlinestart | inlineend)** <expr>
- pragma-msg*: **pragma (message | warning | error)** <expr>
- pragma-verblev*: **pragma verboselevel** <expr>
- pragma-include*: **pragma include** <expr>
- pragma-spansymb*: **pragma spansymbol** <expr>

Description

`pragma` is a special directive. Please note that if ISPP fails to parse parameters of this directive (because of typo or wrong syntax), no error will occur – only a warning will be issued; this is done for compatibility with other preprocessors, which can have their own syntax of `pragma` directive.

First syntax of `pragma` directive controls the options, which ISPP uses to read the source. There are two groups of options. Each group consists of 26 flags (not all of them are meaningful and used by ISPP, though). Each flag has an assigned latin letter. You specify options by typing group name (`option` or `parseroption`), then the letter following the dash. After a letter a plus or minus sign shall be specified. Plus sign to turn the option on, minus to turn it off. Unrestricted number of options can be specified at once (see syntax). The list of options is provided at the end of this topic.

First group of options (`option`) controls the options of the whole ISPP engine, while second group (`parseroption`) controls options specific to parser. The list of options is provided at the end of this topic.

Second syntax is used to specify inline directive terminators: starting and ending, respectively. After the token description keyword (`inlinestart` or `inlineend`) a string type expression must follow. It must not evaluate to an empty string. Only first seven symbols from the string are taken. It is allowed to specify the same token for both starting and ending terminators. By default, `{#` (opening brace and a number sign) and `}` (closing brace) are assumed.

Third syntax of `pragma` directive issues a message of the type specified by the keyword following the directive name. Messages and warnings are sent to the messages window of the compiler. Errors are shown (by the compiler) using message boxes. Expression must be of type string.

Fourth syntax sets the level of verbosity. When the verbose mode is on (see below), this syntax controls the level of importance of messages.

Least important messages will show up only when highest verbose level (9) is set.

Fifth syntax sets the include path. Expression may specify multiple paths delimited with semicolons. The list of these directories is used when ISPP tries to find a file, mentioned in `include` directive.

The last syntax sets the symbol used to span multiple lines together. Expression must not evaluate to an empty string. Only first symbol in string is taken.

ISPP options

- c Indicates that the [translation](#) should be sent to the compiler after preprocessing is done. Default state: on.
- e Specifies whether empty lines from the source should be emitted to the [translation](#), as well as lines with ISPP directives should be replaced with empty lines. Default state: off.
- v Turns on/off the verbose mode. Default state: off.

Parser options

- b Short-circuit boolean evaluation. Default state: on.
- m Short-circuit multiplication evaluation. (In " $0 * A$ ", A will not be evaluated, since the result of expression is known to be zero.)
Default state: off.
- p Pascal-style string literals. In off state uses C-style string literals (with escape sequences). Default state: on.
- u Allow undeclared identifiers. If an undefined identifier is encountered, ISPP will raise an error unless this option is turned on, in which case a standalone identifier (the one that does not look like a function call) will be considered void value. Default state: off.

Examples

```
#pragma parseroption -b- -u+  
#pragma option -c-  
#pragma error "Variable value is: " + MyVar  
#pragma verboselevel 9  
#pragma inlinestart "$("  
#pragma inlineend ")"  
#pragma include __INCLUDE__ + ";D:\INCLUDE"  
#pragma spansymbol "_"
```

Inno Setup Preprocessor: #error

Syntax

error-directive: **error** <text>

Description

Directive causes the Inno Setup compiler to issue an error message with the specified text. Unlike `pragma error`, text in `error` directive is not parsed, so it is recommended to use this directive instead of [pragma](#) when possible to avoid possible syntax errors that may hide real errors your script is trying to report.

Examples

```
#if VER < 0x04000000  
    #error A more recent version of Inno Setup is  
    required to compile this script  
#endif
```

See also

[pragma](#), [if](#).

Inno Setup Preprocessor: GetFileVersion

Prototype

```
str GetFileVersion(str)
```

Description

GetFileVersion function takes a string argument which must be set to the name of the file whose version information is to be queried. The function returns string composed of four decimal numbers delimited with periods. If file does not contain valid version info, the function returns an empty string.

ISPP also has [GetStringFileInfo](#) function, which also can be used to retrieve file version (using "FileVersion" or "ProductVersion" as second parameter). The difference is that GetFileVersion takes it from fixed block of version info, unlike GetStringFileInfo, which extracts string from language specific block.

Inno Setup Preprocessor: GetStringFileInfo

Prototype

```
str GetStringFileInfo(str 1, str 2, int? 3)
```

Description

GetStringFileInfo function retrieves string from specified file's (first argument) version information resource.

Second argument is the name of the version info string-value. This should be one of the predefined strings. Those strings and shortcut macros are typically declared in [ISPPBuiltins.iss file](#).

Third optional argument should specify language and charset identifier. For example: 0x04BE0409 stands for "English (United States)." If this parameter is omitted, ISPP scans for all available version info blocks to find the value.

The function returns an empty string, if it was unable to retrieve the desired string-value.

Inno Setup Preprocessor: Int

Prototype

```
int Int(any 1, int? 2)
```

Description

Function converts an expression (first argument) to its integer representation. If the expression is an integer, the result of the function is the expression value. If the expression is void, the result is 0. If the expression is string, ISPP tries to convert it to integer; if such attempt fails, an error is raised unless second parameter specifies the default result.

Inno Setup Preprocessor: Str

Prototype

`str Str(any)`

Description

Function converts an expression to string. If the expression is integer, the result is its string representation. If the expression is void, the result is an empty string. Otherwise the result is the value of the expression.

Inno Setup Preprocessor: FileExists

Prototype

```
int FileExists(str)
```

Description

Returns non-zero value if the specified file exists.

Inno Setup Preprocessor: FileSize

Prototype

```
int FileSize(str)
```


Description

Returns size, in bytes, of the specified file. If the file does not exist, the result is -1. Beware of ISPP supporting only signed 32 bit integers: for files larger than 2 GB (and smaller than 4 GB) the result is negative.

Inno Setup Preprocessor: ReadIni

Prototype

```
str ReadIni(str 1, str 2, str 3, str? 4)
```

Description

Reads the value from an INI file. Argument 1 must be the name of the INI file, argument 2 – the name of a section in the INI file, the third argument is the key in the section to read. Last optional argument can be used to provide the default value that will be returned on failure, if it is omitted, an empty string is returned.

Inno Setup Preprocessor: WriteIni

Prototype

```
void WriteIni(str 1, str 2, str 3, any 4)
```

Description

Writes specified value to an INI file. Argument 1 is the name of the INI file, argument 2 – the name of a section in the INI file, argument 3 – the name of a key in the section. Last argument should be set to the value you wish to be written to the INI file, it can be of any type.

Inno Setup Preprocessor: ReadReg

Prototype

any ReadReg(**int** 1, **str** 2, **str?** 3, **any?** 4)

Description

Reads the value of the specified key in system registry. First parameter is the root key, such as HKEY_LOCAL_MACHINE. Constants for use with this parameter are typically declared in [ISPPBuiltins.iss file](#) accompanying ISPP. Second parameter specifies a subkey. Third parameter specifies the name of the value, if this parameter is omitted, a default value is assumed. Last optional parameter may be used to specify the default value that will be returned on failure.

Note that this function can return value of any type depending on the type of actual value in registry.

Inno Setup Preprocessor: Exec

Prototype

```
int Exec(str 1, str? 2, str? 3, int? 4, int? 5)
```

Description

Executes specified executable file.

First argument specifies the filename of the module to execute.

Second argument may be used to specify command line to execute.

Third argument may be used to specify the working directory of the process.

Fourth argument should be set to zero, if you don't wish to wait for the process to finish, and non-zero otherwise. By default, non-zero value is assumed.

Fifth argument can be any of the SW_* constants typically defined in [ISPPBuiltins.iss file](#). For GUI processes, it specifies the default value the first time ShowWindow is called. By default, SW_SHOWNORMAL (i. e. 1) is assumed.

If fourth argument is omitted or is non-zero, the function returns the exit code of the process. Otherwise, the function result indicates whether the process has been successfully launched (non-zero for success).

Inno Setup Preprocessor: Copy

Prototype

```
str Copy(str 1, int 2, int? 3)
```

Description

Function extracts a substring from a string (first argument). The 1-based index of the character from which the substring should start is specified by the second argument. The third argument specifies the number of characters to take, if it is omitted, all characters up to the end of the string are copied to the result.

Inno Setup Preprocessor: Pos

Prototype

```
int Pos(str 1, str 2)
```

Description

Pos searches for a substring (first argument) in another string (second argument) and returns an integer value that is the 1-based index of the first character of the substring within the second string. Pos is case-sensitive. If the substring is not found, Pos returns zero.

Inno Setup Preprocessor: RPos

Prototype

```
int RPos(str 1, str 2)
```

Description

RPos searches for a substring (first argument) in another string (second argument) and returns an integer value that is the 1-based index of the first character of the last occurrence of the substring within the second string. RPos is case-sensitive. If the substring is not found, RPos returns zero.

Inno Setup Preprocessor: Len

Prototype

```
int Len(str)
```


Description

Returns the length of the given string.

Inno Setup Preprocessor: SaveToFile

Prototype

```
void SaveToFile(str)
```

Description

This function saves [current translation](#) to the specified file.

Inno Setup Preprocessor: Find

Prototype

```
int Find(int 1, str 2, int? 3, str? 4, int? 5, str?  
6, int? 7)
```

Description

`Find` function is intended to be used with [insert](#) directive. Function returns the index of the line in a [translation](#) depending on specified criteria.

First parameter denotes the index of the line to start the search from, usually it is set to zero.

Second, fourth, and sixth parameters should specify string(s) to search within each line. Only the second parameter must be specified whereas fourth and sixth may be omitted.

Third, fifth, and seventh parameters should specify the search flags for each string meaning that third parameter specifies flags for second, fifth for fourth, and seventh for sixth.

If any of the 'flags' parameters is omitted but the string parameter preceding it is specified, `FIND_MATCH` | `FIND_AND` (i. e. 0) is assumed.

Values for third, fifth, and seventh parameters of `Find` function are typically declared in [ISPPBuiltins.iss file](#). See [Find flags](#) topic for the description of each value.

See also

[insert.](#)

- [Find flags](#)

Inno Setup Preprocessor: SetupSetting

Prototype

```
str SetupSetting(str)
```

Description

SetupSetting function parses [\[Setup\] section](#) in [current translation](#) to find the key whose name is specified as function parameter. Function returns the value of that key if it's found, or an empty string otherwise.

Inno Setup Preprocessor: SetSetupSetting

Prototype

```
void SetSetupSetting(str 1, str 2)
```

Description

SetSetupSetting function modifies or generates [Setup] section directive given the key (first parameter) and its value (second parameter).

If there is no [Setup] section in [current translation](#) (it may happen that function is called before that section in a script), its header (as well as directive itself) is generated by this function.

Please use this function carefully – it should not be called when ISPP is in insert mode (i. e. after [insert](#) directive).

Inno Setup Preprocessor: LowerCase

Prototype

```
str LowerCase(str)
```


Description

LowerCase returns a string with the same text as the string passed in its parameter, but with all letters converted to lowercase. The conversion affects only 7-bit ASCII characters between 'A' and 'Z'.

Inno Setup Preprocessor: EntryCount

Prototype

```
int EntryCount(str)
```

Description

Function returns the total number of entries in specified section in [current translation](#). It does not count empty lines or comments. Function takes care of multiple sections with the same name and counts all of them.

Inno Setup Preprocessor: GetEnv

Prototype

```
str GetEnv(str)
```

Description

Returns the value of the environment variable whose name is specified as the parameter. Returns empty string if variable is not defined.

Inno Setup Preprocessor: DeleteFile

Prototype

```
void DeleteFile(str)
```

Description

Marks file for deletion after compilation is done (no matter successful it was or not). Does not return anything.

Inno Setup Preprocessor: CopyFile

Prototype

```
void CopyFile(str 1, str 2)
```

Description

Copies an existing file (first parameter) to a new file (second parameter). If the new file already exists, it will be overwritten.

Inno Setup Preprocessor: FindFirst

Prototype

```
int FindFirst(str, int)
```

Description

`FindFirst` searches the directory specified by first parameter for the first file that matches the file name implied by first parameter and the attributes specified by second parameter. If the file is found, the result is a find handle, that should be used in subsequent calls to [FindGetFileName](#), [FindNext](#), and [FindClose](#) functions, otherwise the return value is 0.

The first parameter is the directory and file name mask, including wildcard characters. For example, `'.*.*'` specifies all files in the current directory).

The second parameter specifies the special files to include in addition to all normal files. Choose from these file attribute constants typically defined in [ISPPBuiltins.iss file](#) when specifying this parameter:

<code>faReadOnly</code>	Read-only files
<code>faHidden</code>	Hidden files
<code>faSysFile</code>	System files
<code>faVolumeID</code>	Volume ID files
<code>faDirectory</code>	Directory files
<code>faArchive</code>	Archive files
<code>faAnyFile</code>	Any file

Attributes can be combined by OR-ing their constants or values. For example, to search for read-only and hidden files in addition to normal files, pass `faReadOnly | faHidden` as the parameter.

Example

```
#define FindHandle
#define FindResult
#define Mask "*.pas"

#sub ProcessFoundFile
  #define FileName FindGetFileName(FindHandle)
  #if LowerCase(Copy(FileName, 1, 4)) == "ispp"
    FileName: {#FileName}; DestDir: {app}\ispp
  #else
    FileName: {#FileName}; DestDir: {app}
  #endif
#endsub

#for {FindHandle = FindResult = FindFirst(Mask, 0);
FindResult; FindResult = FindNext(FindHandle)}
ProcessFoundFile
#if FindHandle
  FindClose(FindHandle)
#endif
```

See also

[define](#), [sub](#), [if](#).

Inno Setup Preprocessor: FindNext

Prototype

```
int FindNext(int)
```

Description

`FindNext` returns the next entry that matches the name and attributes specified in a previous call to [FindFirst](#). The parameter must be find handle returned by `FindFirst`. The return value is non-zero if the function was successful.

Inno Setup Preprocessor: FindClose

Prototype

```
void FindClose(int)
```

Description

`FindClose` terminates `FindFirst/FindNext` sequence. The parameter must be non-zero find handle returned by `FindFirst`.

This function is obsolete since 1.2. ISPP automatically frees resources allocated in a call to `FindFirst`.

Inno Setup Preprocessor: FindGetFileName

Prototype

```
str FindGetFileName(int)
```

Description

Feed FindGetFileName with the find handle returned by [FindFirst](#) to get the name of the file found by the last call to FindFirst or [FindNext](#).

Inno Setup Preprocessor: FileOpen

Prototype

```
int FileOpen(str)
```

Description

This function opens a text file for reading and returns the file handle (or zero on failure) to be used in subsequent calls to `File*` functions.

Inno Setup Preprocessor: FileRead

Prototype

```
str FileRead(int)
```


Description

The function reads the next line in a text file opened with [FileOpen](#). The only parameter should be the file handle returned by `FileOpen`.

Example

```
#define FileHandle
#define FileLine
#sub ProcessFileLine
    #pragma message FileLine
#endsub
#for {FileHandle = FileOpen("c:\autoexec.bat"); \
    FileHandle && !FileEof(FileHandle); FileLine =
FileRead(FileHandle)} \
    ProcessFileLine
#if FileHandle
    #expr FileClose(FindHandle)
#endif
```

See also

[define](#), [sub](#), [pragma](#), [for](#), [if](#).

Inno Setup Preprocessor: FileReset

Prototype

```
void FileReset(int)
```

Description

The function resets the file pointer to zero, so the subsequent call to [FileRead](#) will read the first line of the file. The only parameter should be the file handle returned by [FileOpen](#).

Inno Setup Preprocessor: FileEof

Prototype

```
int FileEof(int)
```


Description

The function returns zero if the file pointer does not point to the end of the file, or non-zero otherwise. If this function returns non-zero value, subsequent calls to [FileRead](#) will fail. The only parameter should be the file handle returned by [FileOpen](#).

Inno Setup Preprocessor: FileClose

Prototype

```
void FileClose(int)
```

Description

The function closes open file and releases all resources allocated by a call to [FileOpen](#). After calling FileClose, the file handle becomes invalid.

This function is obsolete since 1.2. ISPP automatically frees resources allocated in a call to FileOpen.

Inno Setup Preprocessor: GetDateTimeString

Prototype

```
str GetDateTimeString(str, str, str)
```

Description

The function returns the current date and time as a string using the specified formatting.

The first parameter is the format string. The second and third parameters denote the DateSeparator and TimeSeparator parameters explained below.

The following format specifiers are supported:

d	Displays the day as a number without a leading zero (1-31).
dd	Displays the day as a number with a leading zero (01-31).
ddd	Displays the day as an abbreviation (Sun-Sat).
dddd	Displays the day as a full name (Sunday-Saturday).
dddddd	Displays the date using the system's short date format.
ddddddd	Displays the date using the system's long date format.
m	Displays the month as a number without a leading zero (1-12). If the m specifier immediately follows an h or hh specifier, the minute rather than the month is displayed.
mm	Displays the month as a number with a leading zero (01-12). If the mm specifier immediately follows an h or hh specifier, the minute rather than the month is displayed.
mmm	Displays the month as an abbreviation (Jan-Dec).
mmmm	Displays the month as a full name (January-December).
yy	Displays the year as a two-digit number (00-99).
yyyy	Displays the year as a four-digit number (0000-9999).
h	Displays the hour without a leading zero (0-23).
hh	Displays the hour with a leading zero (00-23).
n	Displays the minute without a leading zero (0-59).
nn	Displays the minute with a leading zero (00-59).
s	Displays the second without a leading zero (0-59).
ss	Displays the second with a leading zero (00-59).
t	Displays the time using the system's short time format.
tt	Displays the time using the system's long time format.
am/pm	Uses the 12-hour clock for the preceding h or hh specifier. Displays 'am' for any hour before noon, and 'pm' for any hour after noon. The am/pm specifier can use lower, upper,

	or mixed case, and the result is displayed accordingly.
a/p	Uses the 12-hour clock for the preceding h or hh specifier. Displays 'a' for any hour before noon, and 'p' for any hour after noon. The a/p specifier can use lower, upper, or mixed case, and the result is displayed accordingly.
/	Displays the date separator character given by the DateSeparator parameter. If DateSeparator is set to an empty string, the system's date separator character will be used instead.
:	Displays the time separator character given by the TimeSeparator parameter. If TimeSeparator is set to an empty string, the system's time separator character will be used instead.
'xx'/"xx"	Characters enclosed in single or double quotes are displayed as-is, and do not affect formatting.

Format specifiers may be written in upper case as well as in lower case letters--both produce the same result.

Example

```
#define MyDateTimeString GetDateTimeString('dddd',  
    '', '');  
#define MyDateTimeString GetDateTimeString('dddd  
    tt', '', '');  
#define MyDateTimeString  
    GetDateTimeString('dd/mm/yyyy hh:nn:ss', '-', ':');
```

Inno Setup Preprocessor: GetFileDateTimeString

Prototype

```
str GetFileDateTimeString(str, str, str, str)
```

Description

The function returns the date and time of the specified file as a string using the specified formatting.

The first parameter is the file name. The second, third and fourth parameters denote the format string, DateSeparator and TimeSeparator parameters as explained in the [GetDateTimeString](#) topic.

Example

```
#define MyFileDateTimeString  
GetFileDateTimeString('myfile.txt', 'dd/mm/yyyy  
hh:nn:ss', '-', ':');
```

Inno Setup Preprocessor: GetMD5OfString

Prototype

```
str GetMD5OfString(str)
```

Description

Gets the MD5 sum of the specified ANSI string, as a string.

Example

```
#define MD5 GetMD5OfString('Test')  
// MD5 = '0cbc6611f5540bd0809a388dc95a615b'
```

Inno Setup Preprocessor: GetMD5OfUnicodeString

Prototype

```
str GetMD5OfString(str)
```

Description

Gets the MD5 sum of the specified Unicode string, as a string.

Causes an internal error if called during non Unicode compilation.

Example

```
#define MD5 GetMD5ofUnicodeString('Test')  
// MD5 = '8e06915d5f5d4f8754f51892d884c477'
```

Inno Setup Preprocessor: GetMD5OfFile

Prototype

```
str GetMD5OfFile(str)
```

Description

Gets the MD5 sum of the specified file, as a string.

Inno Setup Preprocessor: GetSHA1OfString

Prototype

```
str GetSHA10fString(str)
```

Description

Gets the SHA-1 hash of the specified ANSI string, as a string.

Inno Setup Preprocessor: GetSHA1OfUnicodeString

Prototype

```
str GetSHA10fString(str)
```

Description

Gets the SHA-1 hash of the specified Unicode string, as a string.

Causes an internal error if called during non Unicode compilation.

Inno Setup Preprocessor: GetSHA1OfFile

Prototype

```
str GetSHA10fFile(str)
```


Description

Gets the SHA-1 hash of the specified file, as a string.

Inno Setup Preprocessor: Trim

Prototype

```
str Trim(str)
```

Description

Returns a copy of the specified string without leading and trailing spaces.

Inno Setup Preprocessor: StringChange

Prototype

```
str StringChange(str, str, str)
```

Description

Returns a copy of the first string, with all occurrences of the second string changed to the third string.

Example

```
#define MyString "a ca c"  
#define MyString2 StringChange(MyString, " ", "b")  
// MyString2 = 'abcabc'
```


Inno Setup Preprocessor: Defined

Prototype

```
int Defined(<ident>)
```

```
int Defined <ident>
```

Description

Special function. It takes an identifier as opposed to an expression.
Returns non-zero if specified identifier is defined with [define](#) directive.

It is allowed to not use parentheses with this function.

Inno Setup Preprocessor: TypeOf

Prototype

```
int TypeOf(<ident>)
```

```
int TypeOf <ident>
```

Description

Special function. It takes an identifier as opposed to an expression. Returns one of predefined values, each of which corresponds to a particular value type (void, integer, or string), if an identifier is a variable name, or identifier type (macro or function) otherwise. Values that this function returns are typically declared in [ISPPBuiltins.iss file](#).

It is allowed to not use parentheses with this function.

Inno Setup Preprocessor: Macros' Local array

In context of macro expression additional array named `Local` is valid. Its elements can be used for temporary storage and reusing values in sequential expressions. This array belongs to one context of macro call, that means that values stored in `Local` array are neither preserved from call to call (including recursive), nor are they accessible from anywhere except the macro expression.

```
#define DeleteToFirstPeriod(str *S) /* macro
declaration */ \
    Local[1] = Copy(S, 1, (Local[0] = Pos(".", S)) -
1), \
    S = Copy(S, Local[0] + 1), \
    Local[1]
```

Inno Setup Preprocessor: Find flags

Description

One of the following four values must be specified:

`FIND_MATCH` (0) means that the line must match the search string.

`FIND_BEGINS` (1) means that the line must start with the search string.

`FIND_ENDS` (2) means that the line must end with the search string.

`FIND_CONTAINS` (3) means that the line must contain (i. e. it also can match, begin, or end with) the search string.

Any of the following modifiers may be combined with one of the previous using bitwise OR operator (`|`):

`FIND_CASESENSITIVE` (4) means that comparison must be case-sensitive.

`FIND_AND` (0) means that this criterium (the pair of arguments in `Find` function) must be met as well as previous criteria.

`FIND_OR` (8) means that it is allowed that this criterium is tested even if previous criteria were not met.

`FIND_NOT` (16) means that this criterium must not be met.

`FIND_AND` and `FIND_OR` are mutually exclusive. If both are specified, `FIND_OR` takes precedence.

Special flags:

`FIND_TRIM` (32) means that leading and trailing whitespaces must be stripped off from the line prior to testing it against all the criteria. This flag can only be used in the third argument of the `Find` function. It is not mutually exclusive with any of the previously mentioned flags.