Inno Setup Preprocessor

Inno Setup Preprocessor (ISPP) is an preprocessor add-on for Inno Setup. ISPP does not depend on any particular version of these products. The only requirement is that version must be 1.3 or higher.

Inno Setup Preprocessor works at compile-time. It has nothing to do with the behaviour of setups.

The main purpose of ISPP is to automate some pre or post compilation tasks and decrease the probability of typos in your scripts. For example, you can declare 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 occurences 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 (for example, in [Code] section). Without ISPP, you would have to modify your script each time version of your application changes.

Conditional compilation (inclusion of parts of the script) is possible with ISPP, therefore, you can maintain only one script for different versions/levels of your installation packages (trial and fully functional, lite and enterpise, etc.).

Topics in this section

- Documentation conventions
- Directives
- <u>Visibility of identifiers</u>
- User defined macros
- Expressions
- Built-in functions
- <u>Predefined and internally defined variables</u>
- BUILTINS.ISS file

• Extended Command Line Compiler

Documentation conventions

Directive syntax documenting conventions

Directive usage syntax in this reference material 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></token>	Non-terminal. Its syntax is either shown before, or explained.

Built-in functions prototypes documenting conventions

Built-in functions prototypes are shown as function result type, function name and list of formal arguments in parentheses. If function has one argument, only the type of the argument is mentioned. Otherwise after the type of the argument follows an integer number for referencing this parameter later in the description of the function.

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.

Directives

In ISPP directives can be used in two ways: simple (standard), or inline.

Simple directives occupy a whole line and must begin with # symbol. A whitespace is allowed before and after that symbol. The name of the directive follows.

Inline directives can appear inside the text line. They must begin and end with defined tokens. Those tokens are contolled via <u>pragma</u> directive. By default {# and } are assumed as starting and terminating tokens, respectively.

Please note that conditional blocks cannot be mixed, i. e. a conditional block can't be started by inline directive and finished by simple one, and vice versa.

See also

Documentation conventions.

Topics in this section

- define directive
- dim directive
- <u>undef directive</u>
- include directive
- file directive
- emit directive
- expr directive
- insert directive
- append directive
- if, elif, else and endif directives
- ifdef, ifndef, ifexsist, ifnexist directives
- for directive
- sub, endsub directives
- pragma directive
- error directive

Visibility of identifiers

Variable (as well as macro) declarations can be defined as "public," "protected," or "private." To define a specifically public, protected, or private variable, its name in <u>define</u> directive should be prepended with one of the keywords (visibility resolution clauses):

```
#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.

Hereinafter in this topic, "variables" also refers to macros as well.

Public variables are ordinary variables as they used to be in previous versions of ISPP. They are accessible (unless any protected or private identifiers override them, see below) from anywhere after the point they are declared.

Protected variables are accessible only in the file they are declared in (declaration file) and in files included by declaration file via <u>include</u> or <u>file</u> directives. They can be said (with some reserves) that they are public variables which are automatically undefined once the declaration 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 explicity 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 <u>include</u> 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, therefore, as in any programming language 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 <u>define</u> directive.

#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 explicity specifying its visibility. When default visibility is not set, public is assumed by default. Note, that 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 parent file doesn't, declare it as protected. Declare it as public otherwise. If you're unsure, then protected visibility is the common case.

User defined macros

You also can define so called "macros" in your scripts. Those macros can be said "user defined functions." ISPP doesn't treat macros the way standard (C or C++) preprocessor does. Please do not try to use C macros in ISS scripts.

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 <u>define</u>.

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 I-values (in other words, other defined variables or expressions that evaluate to I-values). If macro expression modifies byreference 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.

Topics in this section

• Macros' Local array

Expressions

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 (therefore, never calls 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 list of operators and their meaning can be found at <u>http://msdn.microsoft.com</u> .

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 is certain cases. For example, in conditional expression, "true" operand can be of string type, whereas "false" operand can be of integer type. This is unacceptable by compiler, but is allowed in interpreter. However, it is not recommended to compose such expressions.
- 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).

Data types

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

If "allow undeclared identifiers" parser option is off (the default state, see <u>pragma</u>), 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 built-in 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 paretheses:

```
#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 delimitter.

Built-in functions

Built-in functions are provided by ISPP itself and are always available in scripts. Unlike macros, built-in functions do not have named arguments, therefore alternative call syntax (using argument names) may not be used. Result type is always the same unless mentioned otherwise (e. g. <u>ReadReg</u> function).

See also

Documentation conventions.

Topics in this section

- Defined function
- <u>TypeOf function</u>
- GetFileVersion function
- <u>GetStringFileInfo function</u>
- Int function
- Str function
- <u>FileExists function</u>
- <u>FileSize function</u>
- <u>ReadIni function</u>
- WriteIni function
- <u>ReadReg function</u>
- Exec function
- <u>Copy function</u>
- Pos function
- <u>RPos function</u>
- Len function
- <u>SaveToFile function</u>
- Find function
- <u>SetupSetting function</u>
- <u>SetSetupSetting function</u>
- LowerCase function
- EntryCount function
- GetEnv function
- DeleteFile function
- <u>CopyFile function</u>
- <u>FindFirst function</u>
- FindNext function
- <u>FindClose function</u>
- FindGetFileName function

- FileOpen function
- FileRead function
- <u>FileReset function</u>
- FileEof function
- <u>FileClose function</u>
- GetDateTimeString function

Predefined and internally defined variables

There are number of predefined variables in ISPP which basically supported for compatibility with other preprocessors. Predefined variables cannot be undefined or redefined. Some of them have special purpose and their value can vary depending on certain circumstances. Depending on the nature of a predefined varialbe, redefinition (or overriding) of it can produce unpredictable results. It is not recommended to do so.

COUNTER	int . Automatically increments each time it is used (afterwards).
FILE	str. Returns name of the current file. Empty
INCLUDE	string for the root file. str . Returns current include path (or paths delimitted with semicolons) set via #pragma include.
LINE	int . Returns number of a line in the current file (not a <u>translation</u>), on which the variable is used (or macro that uses this variable is called).
0PT_X	void . Defined if specified option set via #pragma option -x+ is in effect. In place of "X" may be any letter from "A" to "Z." Use <u>Defined</u> function to test whether the variable
PATHFILENAME	is defined. str . Similar toFILE but, returns full pathname of the file. Empty string for the root file.
POPT_X	void . Defined if specified parser option set via #pragma parseroption -x+ is in effect. In place of "X" may be any letter from "A" to "Z." Use <u>Defined</u> function to test whether the variable is defined.
WIN32	void . Always defined.
ISPP_INVOKED	void. Always defined.
PREPROCVER	int . Returns 32-bit encoded version of ISPP. Highest byte holds major version, lowest byte

holds the build number. **void**. Always defined.

Internally ISPP defines some special variables. These variables are usual <u>public</u> variables with the only difference that their declaration is handled by ISPP automatically. You can redefine or undefine them.

WINDOWS

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

BUILTINS.ISS file

The BUILTINS.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 <u>include</u> directive. This file contains common declarations, such as special constants for using with some built-in 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.

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 <*script name*> [options]. Or to read from standard input: iscc - [options].

Options are to emulate a <u>define</u> or <u>pragma</u> directive are:

/d <name>[=<value>]</value></name>	Sets#define public <name> <value></value></name>
/\$ <letter>(+ -)</letter>	Sets#pragma option - <letter> (+ -)</letter>
/p <letter>(+ -)</letter>	Sets#pragma parseroption - <letter>(+ -)</letter>
/i <paths></paths>	Sets#pragma include - <paths></paths>
/s <string></string>	Sets#pragma inlinestart - <string></string>
/e <string></string>	<pre>Sets #pragma inlineend -<string></string></pre>
/v <number></number>	Sets#pragma verboselevel - <number></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:\isetup\samples\my script.iss" /\$c-/pu+ "/dLic=Trial Lic.txt" /iC:\INC;D:\INC

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.

pragma directive

Syntax

pragma-directive:	<pragma-option> <pragma-itokens> <pragma-msg> <pragma-verblev> <pragma-include> <pragma-spansymb></pragma-spansymb></pragma-include></pragma-verblev></pragma-msg></pragma-itokens></pragma-option>
pragma-option:	<pre>pragma (option parseroption) - <letter> (+ -) [- <letter> (+ -)]</letter></letter></pre>
pramga-itokens:	pragma (inlinestart inlineend) <expr></expr>
pragma-msg:	pragma (message warning error) <expr></expr>
pragma-verblev:	pragma verboselevel <expr></expr>
pragma-include:	pragma include <expr></expr>
pragma-spansym	b: pragma spansymbol <expr></expr>

Description

pragma is a special directive. Please note that if ISPP fails to parse parameters of this directive (because of typo or wrong sytax), no error will occur – only a warning will be issued; this is done for compatibility with other preprocessors, which can have their own sytax 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 meaningfull 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 delimitted with semicolons. The list of these directories is used when ISPP tries to find a file, mentioned in <u>include</u> 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 <u>translation</u> 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 <u>translation</u>, 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 "_"
```

define directive

Syntax

define-directive:	<default-visibility-set> <variable-definition> <macro-definition></macro-definition></variable-definition></default-visibility-set>			
default-visibility-set.	(define :) private protected public			
variable-definition:	(define :) [private protected public] <ident> [[<expr>]] [[=] <expr>]</expr></expr></ident>			
macro-definition:	(define :) [private protected public] <ident> ([<formal-macro-args>]) <expr></expr></formal-macro-args></ident>			
formal-macro-args:	<formal-macro-arg> [, <formal-macro-arg>]</formal-macro-arg></formal-macro-arg>			
formal-macro-arg:	<by-val-arg> <by-ref-arg></by-ref-arg></by-val-arg>			
<i>by-val-arg:</i> [<type-id>] <ident> [= <expr>]</expr></ident></type-id>				
<i>by-ref-arg:</i> [<typ< td=""><td>pe-id>] * <ident></ident></td></typ<>	pe-id>] * <ident></ident>			
type-id: any int str func				

Description

The first syntax of the define directive 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.

The second syntax 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 which is set by the first syntax of define directive.

The third syntax 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.

Examples

#define MyAppName "My Program" ; define variable
#define MyAppVer GetFileVersion("MyProg.exe") ;
define variable
#define MyArray[0] 15 ; assign a value to an array
element
#define Multiply(int A, int B = 10) A * B ; define
macro

See also

dim, undef, User defined macros, Visibility of identifiers.

dim directive

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

Description

Declares and sets dimension of an array variable. All elements of an array are initialized to null (void). To assign a value to an element, use <u>define</u> directive specifying array index in square brackets after array idetifier. It is also possible to use array elements as the left operand of assignment operators, it is not necessary to <u>define</u> them first.

If the default visibility (see first syntax of <u>define</u>) differs from the one specified in dim directive, it must be specified in <u>define</u> as well, when assigning a value to an element.

Prior to using array, it must be declared with this directive. You cannot apply <u>define</u> directive on an element of an array that has not been defined yet.

Examples

#dim MyArray[10] ; declare an array of 10 elements

See also

define, undef, Visibility of identifiers.

undef directive

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

Description

Removes variable or macro from the list. If no <u>visibility</u> resolution clause (public, protected, or private) is specified, ISPP first tries to remove private (to this file) variable, if there is no private (to this file) variable of the given name, it will try to remove protected and so on. When visibility resolution clause is specified, only the variable from that visibility (in case of protected the one that was declared earlier and closer to the point undef was used) gets deleted.

Examples

#undef MyVar #undef MyMacro #undef public MyVar

See also

define, dim, Visibility of identifiers.

include directive

include-directive: (include | +) < <filename> >
 (include | +) <expr>

Description

Includes contents of the file preprocessing it.

If filename is enclosed in angle brackets, preprocessor 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 <u>pragma</u> includepath, then on path specified by INCLUDE environment variable.

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

This directive cannot be inline.

Examples

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

See also

<u>file, sub</u>.

file directive

file-directive: file <expr>

Description

Replaces the directive with the name of a temporary file containing <u>translation</u> of the specified file. Upon end of compilation, this file is deleted. Including a file using this directive creates a new independent instance of preprocessor object, passing it options currently in effect and all declared identifiers. Note that if included file modifies options in some way, they are not propagated back to current preprocessor object.

When using this directive in the Source parameter of the [Files] section, specify DestName – temporary file name does not match the original name.

This directive can be only inline.

Examples

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

See also

<u>include</u>.

emit directive

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

Description

Replaces the directive with the value of expr.

This directive is likely to be used as inline. In this case, its name can be omitted; however, if an expression to be emitted to <u>translation</u> begins with an ambiguous identifier that may be treated as directive name, emit or its shorthand symbol (=) must be specified.

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

<u>expr</u>.

expr directive

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

Description

Evaluates an expression ignoring its result. This directive acts like <u>emit</u> with the exception that it doesn't emit anything to the <u>translation</u>.

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

Examples

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

See also

<u>emit</u>.

insert directive

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

Description

Changes the insertion point. By default, each line is processed and added to the end of the <u>translation</u>. With <u>insert</u> directive the point at which the translated line is added to the <u>translation</u> can be changed. The directive takes an expression that must evaluate to an integer. The insertion point (specifically: index of a line in <u>translation</u> before which subsequent lines will be inserted) will be set to the number to which expression evaluates. All lines after this directive will be inserted at that insertion point. Insertion point is automatically incremented each time a line is added to the <u>translation</u>, so that each line is inserted after the one previously inserted.

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

<u>Find</u> built-in function can be used to produce value for the insert directive.

Examples

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

See also

<u>append</u>.

append directive

append-directive: append

Resets the insertion point (previously set by <u>insert</u> directive) to the end of <u>translation</u>. All subsequent lines will be appended to existing <u>translation</u>.

See also

<u>insert</u>.

if, elif, else and endif directives

Syntax

if-directive:	if <expr></expr>
elif-directive:	elif <expr></expr>
else-directive:	else
endif-directive:	endif

The if directive, with the elif, else, and endif directives, controls compilation of portions of a source file. If the expression you write (after the if) has a nonzero value, the line group immediately following the if directive is retained in the translation unit.

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 nest in the text portions of other if directives. Each nested else, elif, or endif directive belongs to the closest preceding if directive.

All conditional-compilation directives, such as if and <u>ifdef</u>, must be matched with closing endif directives prior to the end of file; otherwise, an error message is generated. When conditional-compilation directives are contained in include files, they must satisfy the same conditions: There must be no unmatched conditional-compilation directives at the end of the include file.

The preprocessor selects one of the given occurrences of text for further processing. A block specified in text can be any sequence of text. It can occupy more than one line. Usually text is program text that has meaning to the compiler or the preprocessor.

The preprocessor processes the selected text and passes it to the compiler. If text contains preprocessor directives, the preprocessor carries out those directives. Only text blocks selected by the preprocessor are compiled.

The preprocessor selects a single text item by evaluating the expression following each if or elif directive until it finds a true (nonzero) constant expression. It selects all text (including other preprocessor directives) up to its associated elif, else, or endif.

If all occurrences of constant-expression are false, or if no elif

directives appear, the preprocessor selects the text block after the else clause. If the else clause is omitted and all instances of constant-expression in the if block are false, no text block is selected.

Inline conditional directives must not be mixed with simple. If the if directive is simple (it occupies one line), its associated directives (elif, else, or endif) must also be simple, as opposed to inline (the directives that are embedded in text 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
```

ifdef, ifndef, ifexsist, ifnexist directives

Syntax

ifdef-directive:	ifdef <ident></ident>
ifndef-directive:	ifndef <ident></ident>
ifexist-directive:	ifexist <expr></expr>
ifnexist-directive:	ifnexist <expr></expr>

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

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 Enterpise
  Filename: "extra.dll"; DestDir: {app}
#endif
```

See also

FileExists function

for directive

Syntax

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

Use for directive to get loop behaviour. for takes 4 expressions, each of which has its own purpose and term. First expression (expr1) is called "initialization expression," second expression (expr2) is called "condition," third expression (expr3) is "action," and the last expression outside braces (expr4) is "loop body."

The logic 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 paragraph 2.

Examples

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

More examples

FindFirst function, FileRead function.

See also

<u>sub, include</u>.

sub, endsub directives

Syntax

sub-directive: sub <ident>
endsub-directive: endsub

sub and endsub directives are used to declare a user defined procedure. Procedure is a closed sequence of ISPP directives and/or script lines, which may be included later once or several times. As of its logic, a user defined procedure is similar to an external file, and a call to a user defined procedure is similar to an inclusion of an external file.

However, procedures can be called from within expressions, whereas inclusion of a file requires a whole directive to be used. Please note, that it is strongly not recommended to call procedures that emit several lines to <u>translation</u> from 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 untill it is called, so if any errors exist (invalid directives, syntax errors) in the 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 function, FileRead function.

See also

<u>User defined macros, if, emit</u>.

error directive

Syntax

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

Directive causes 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 <u>pragma</u> 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</pre>

See also

<u>pragma, if</u>.

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]
```

Defined function

Prototype

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

Special function. It takes an identifier as opposed to an expression. Returns non-zero if specified identifier is defined with <u>define</u> directive.

It is allowed to not use parentheses with this function.

ReadReg function

Prototype

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

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 <u>BUILTINS.ISS file</u> 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.

TypeOf function

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

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 <u>BUILTINS.ISS file</u>.

It is allowed to not use parentheses with this function.

GetFileVersion function

str GetFileVersion(str)

GetFileVersion function takes 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 delimitted with periods. If file does not contain valid version info, the function returns an empty string.

ISPP also has <u>GetStringFileInfo</u> 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.

GetStringFileInfo function

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

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 <u>BUILTINS.ISS file</u>.

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.

Int function

int Int(any 1, int? 2)

Function converts an expression 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 value.

Str function

str Str(any)

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.

FileExists function

int FileExists(str)

Returns non-zero value if the file (whose name specified as parameter) exists.

FileSize function

int FileSize(str)

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.

ReadIni function

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

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.

WriteIni function

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

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.

Function does not return any significiant value.

Exec function

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

Executes specified executable file.

First argument shall specify 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 <u>BUILTINS.ISS file</u>. 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).

Copy function

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

Function extracts a substring from the first string type parameter. The 1-based index of the character from which the substring should start is specified in 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.

Pos function

int Pos(str 1, str 2)

Pos searches for first parameter within second parameter and returns an integer value that is the index of the first character of first string within second. Pos is case-sensitive. If substring is not found, Pos returns zero.

RPos function

int RPos(str 1, str 2)

RPos searches for first parameter within second parameter and returns an integer value that is the index of the first character of last occurence of the first string within second. RPos is case-sensitive. If substring is not found, RPos returns zero.

Len function

int Len(str)

Returns the length of the given string.

SaveToFile function

void SaveToFile(str)

This function saves <u>current translation</u> to the specified file. Function does not return any significiant value.

Find function

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

Find function is intended to be used with <u>insert</u> directive. Function returns the index of the line in a <u>translation</u> 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 <u>BUILTINS.ISS file</u>. See <u>Find flags</u> topic for the description of each value.

See also

<u>insert</u>.

Topics in this section

• Find flags

SetupSetting function

str SetupSetting(str)

SetupSetting function parses [Setup] section in <u>current translation</u> 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.

SetSetupSetting function

void SetSetupSetting(str 1, str 2)

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 <u>current translation</u> (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 <u>insert</u> directive).

The function does not return any significiant value.

LowerCase function

str LowerCase(str)

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'.

EntryCount function

int EntryCount(str)

Function returns the total number of entries in specified section in <u>current translation</u>. It does not count empty lines or comments. Function takes care of multiple sections with the same name and counts all of them.

GetEnv function

str GetEnv(str)

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

DeleteFile function

void DeleteFile(str)

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

CopyFile function

void CopyFile(str 1, str 2)

Copies a file.

FindFirst function

int FindFirst(str, int)

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 function functions, otherwise the return value is 0.

The first parameter is the directory and file name mask, including wildcard characters. For example, $' \cdot \lambda^* \cdot *'$ 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 <u>BUILTINS.ISS file</u> when specifying this parameter:

faReadOnly	Read-only files
faHidden	Hidden files
faSysFile	System files
faVolumeID	Volume ID files
faDirectory	Directory files
faArchive	Archive files
faAnyFile	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

<u>define, sub, if</u>.

FindNext function

int FindNext(int)

FindNext returns the next entry that matches the name and attributes specified in a previous call to <u>FindFirst</u>. The parameter must be find handle returned by FindFirst. The return value is non-zero if the function was successful.

FindClose function

void FindClose(int)

FindClose termiates <u>FindFirst/FindNext</u> 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.

FindGetFileName function

str FindGetFileName(int)

Feed FindGetFileName with the find handle returned by <u>FindFirst</u> to get the name of the file found by the last call to FindFirst or <u>FindNext</u>.

FileOpen function

int FileOpen(str)

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.

FileRead function

str FileRead(int)

The function reads the next line in a text file opened with <u>FileOpen</u>. The only parameter should be the file handle returned by FileOpen.

Example

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

See also

<u>define</u>, <u>sub</u>, <u>pragma</u>, <u>for</u>, <u>if</u>.

FileReset function

void FileReset(int)

The function resets the file pointer to zero, so the sugsequent call to <u>FileRead</u> will read the first line of the file. The only parameter should be the file handle returned by <u>FileOpen</u>.

FileEof function

int FileEof(int)

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 <u>FileRead</u> will fail. The only parameter should be the file handle returned by <u>FileOpen</u>.

FileClose function

void FileClose(int)

The function closes open file and releases all resources allocated by a call to <u>FileOpen</u>. 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.

GetDateTimeString function

str GetDateTimeString(str, str, str)

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).
ddddd	Displays the date using the system's short date format.
ddddd	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).
уу	Displays the year as a two-digit number (00-99).
уууу	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 a rea / a rea	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,

a/p	or mixed case, and the result is displayed accordingly. Uses the 12-hour clock for the preceding h or hh specifier. Displays 'a' for any hour before noon, and 'p' for any hour
1	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
'xx'/"xx"	empty string, the system's time separator character will be used instead. 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('ddddd',
'', '');
#define MyDateTimeString GetDateTimeString('ddddd
tt', '', '');
#define MyDateTimeString
GetDateTimeString('dd/mm/yyyy hh:nn:ss', '-', ':');
```

Translation refers to the preprocessed script.

Current <u>translation</u> refers to current output of ISPP, the translated (preprocessed) part of the script up to the point (or line) which ISPP is currently processing.

Find flags

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 casesensitive.

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 previos 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.