What is Inno Setup?

Inno Setup version 5.5.6 Copyright © 1997-2015 Jordan Russell. All rights reserved. Portions Copyright © 2000-2015 Martijn Laan. All rights reserved. Inno Setup home page

Inno Setup is a *free* installer for Windows programs. First introduced in 1997, Inno Setup today rivals and even surpasses many commercial installers in feature set and stability.

Key features:

- Support for every Windows release since 2000, including: Windows 10, Windows 8, Windows Server 2012, Windows 7, Windows Server 2008 R2, Windows Vista, Windows Server 2008, Windows XP, Windows Server 2003, and Windows 2000. (No service packs are required.)
- Extensive support for installation of <u>64-bit</u> applications on the 64-bit editions of Windows. Both the x64 and Itanium architectures are supported. (On the Itanium architecture, Service Pack 1 or later is required on Windows Server 2003 to install in 64-bit mode.)
- Supports creation of a single EXE to install your program for easy online distribution. <u>Disk spanning</u> is also supported.
- Standard Windows wizard interface.
- Customizable setup types, e.g. Full, Minimal, Custom.
- Complete <u>uninstall</u> capabilities.
- Installation of <u>files</u>:

Includes integrated support for "deflate", bzip2, and 7-Zip LZMA/LZMA2 file <u>compression</u>. The installer has the ability to compare file version info, replace in-use files, use shared file counting, register DLL/OCX's and type libraries, and install fonts.

• Creation of shortcuts anywhere, including in the Start Menu and on

the desktop.

- Creation of <u>registry</u> and <u>.INI</u> entries.
- <u>Running</u> other programs before, during or after install.
- Support for <u>multilingual</u> installs, including right-to-left language support.
- Support for <u>passworded</u> and <u>encrypted</u> installs.
- Support for <u>digitally signed</u> installs and uninstalls.
- Silent install and silent uninstall.
- Unicode installs.
- Integrated preprocessor option for advanced compile-time customization.
- Integrated <u>Pascal scripting</u> engine option for advanced run-time install and uninstall customization.
- Full source code is available (Borland Delphi 2.0-5.0 and 2009).

Is it really free of charge, even for commercial use?

Yes, it may be used completely free of charge, even when deploying commercial applications.

(Note: "Completely free of charge" must not be confused with "completely free". Inno Setup is copyrighted software, *not* public domain software. There are some restrictions on distribution and use; see the LICENSE.TXT file for details.)

Documentation Conventions

monospaced text When you see monospaced text in the documentation, it refers to text you would type in a <u>script</u> file.

Creating Installations

Installations are created by means of *scripts*, which are ASCII text files with a format somewhat similar to .INI files. (No, it's not as complicated as you might be thinking!). <u>Unicode Inno Setup</u> also supports UTF-8 encoded text files.

Scripts have an ".iss" (meaning Inno Setup Script) extension. The script controls every aspect of the installation. It specifies which files are to be installed and where, what shortcuts are to be created and what they are to be named, and so on.

Script files are usually edited from inside the Setup Compiler program. After you have finishing writing the script, the next and final step is select "Compile" in the Setup Compiler. What this does is create a complete, ready-to-run Setup program based on your script. By default, this is created in a directory named "Output" under the directory containing the script.

To give you an idea of how this all works, start the Setup Compiler, click *File* | *Open*, and select one of the script files in the Examples subdirectory located under the Inno Setup directory. (It may be helpful to use the sample scripts as a template for your own scripts.)

See also:

Script Format Overview

Script Format Overview

Inno Setup Scripts are arranged into *sections*. Each section controls a different aspect of the installation. A section is started by specifying the name of the section enclosed in square brackets []. Inside each section is any number of *entries*.

There are two different main types of sections: those such as [Setup] whose entries contain directive names and values (in the form Directive=Value), and those such as [Files] whose entries are divided into parameters.

Here is an example:

```
[Setup]
AppName=My Program
[Files]
Source: "MYPROG.EXE"; DestDir: "{app}"
```

Note that it is legal to specify multiple sections of the same name.

You can put "comments" in the script (which are ignored by the compiler) by placing a semicolon at the beginning of a line. For example:

```
; This is a comment. I could put reminders to mys
```

A C-like #include directive is supported, which pulls in lines from a separate file into the script at the position of the #include directive. The syntax is:

```
#include "filename.txt"
```

If the filename is not fully qualified, the compiler will look for it in the same directory as the file containing the #include directive. The filename may be prefixed by "compiler:", in which case it looks for the file in the Compiler directory.

A #preproc directive is supported, which specifies whether to use the

built-in preprocessor which only supports the above #include directive or to use Inno Setup Preprocessor (ISPP) which supports many more directives. The syntax is:

#preproc builtin
#preproc ispp

By default, scripts use ISPP if available, and .isl files use the built-in preprocessor.

See also: Parameters in Sections Constants **Common Parameters Components and Tasks Parameters** [Setup] section [Types] section [Components] section [Tasks] section [Dirs] section [Files] section [Icons] section [INI] section [InstallDelete] section [Languages] section [Messages] section [CustomMessages] section [LangOptions] section [Registry] section [Run] section [UninstallDelete] section [UninstallRun] section Pascal Scripting: Introduction

Parameters in Sections

All of the sections in a script, with the exception of [Setup], [Messages], [CustomMessages], [LangOptions], and [Code], contain lines separated into *parameters*. The following is an example of a [Files] section:

```
[Files]
Source: "MYPROG.EXE"; DestDir: "{app}"
Source: "MYPROG.CHM"; DestDir: "{app}"
Source: "README.TXT"; DestDir: "{app}"; Flags: is
```

Each parameter consists of a name, followed by a colon, and then a value. Unless otherwise noted, parameters are optional in that they assume a default value if they are not specified. Multiple parameters on a line are separated by semicolons, and can be listed in any order.

The value of a parameter is traditionally surrounded in double quotes (") when it contains a user-defined string, such as a filename. Using quotes is not required, though, but by doing so it makes it possible to embed leading and trailing spaces in the value, as well as semicolons and double-quote characters.

To embed a double-quote character inside a quoted value, use two consecutive double-quote characters. For example:

```
"This "" contains "" embedded "" quotes"
```

The Setup Compiler would see that as:

```
This " contains " embedded " quotes
```

If you want the value of a parameter to be a single double-quote character, use four double-quote characters: """". The outer two are needed to surround the string in quotes; the inner two are used to embed a single double-quote character.

Constants

The majority of the script entries can have *constants* embedded in them. These are predefined strings enclosed in brace characters { }. Setup or Uninstall translates the constants to their literal values, depending on the user's choices and system configuration. For example, {win}, as described below, would translate to "C:\WINDOWS" on most systems.

A "{" character is treated as the start of the constant. If you want to use that actual character in a place where constants are supported, you must use two consecutive "{" characters. (You do not need to double "}" characters.)

When a backslash immediately follows a constant, Setup or Uninstall will automatically remove the backslash if the value of the constant ends in a backslash already. Thus, if the value of a particular constant is "C:\", {constantname}\file will translate to "C:\file", not "C:\file". If you want to prevent this from happening, enclose the backslash in { } characters, e.g. {app}{\}.

The following is the list of supported constants.

Directory Constants

{app}

The application directory, which the user selects on the *Select Destination Location* page of the wizard.

For example: If you used {app}\MYPROG.EXE on an entry and the user selected "C:\MYPROG" as the application directory, Setup will translate it to "C:\MYPROG\MYPROG.EXE".

{win}

The system's Windows directory.

For example: If you used {win}\MYPROG.INI on an entry and the system's Windows directory is "C:\WINDOWS", Setup or Uninstall will translate it to "C:\WINDOWS\MYPROG.INI".

{sys}

The system's System32 directory.

For example: If you used {sys}\CTL3D32.DLL on an entry and the system's Windows System directory is

"C:\WINDOWS\SYSTEM", Setup or Uninstall will translate it to "C:\WINDOWS\SYSTEM\CTL3D32.DLL".

On 64-bit Windows, by default, the System32 path returned by this constant maps to the directory containing 32-bit system files, just like on 32-bit Windows. (This can be overridden by enabling <u>64-bit mode</u>.)

{syswow64}

On 64-bit Windows, the system's SysWOW64 directory, typically "C:\WINDOWS\SysWOW64". This is the actual directory in which 32-bit system files reside. On 32-bit Windows, 32-bit system files reside in "System32" or "System", not in a separate SysWOW64 directory, so this constant will resolve to the same directory as {sys} if used there.

Do not use this constant unless you have a specific need to obtain the name of the actual directory in which 32-bit system files reside. Gratuitously using {syswow64} in places where {sys} will suffice may cause problems. (See the documentation for the [Files] section's sharedfile flag for one example.)

{src}

The directory in which the Setup files are located. For example: If you used {src}\MYPROG.EXE on an entry and the user is installing from "S:\", Setup will translate it to "S:\MYPROG.EXE".

{sd}

System Drive. The drive Windows is installed on, typically "C:". This directory constant is equivalent to the *SystemDrive* environment variable.

{pf}

Program Files. The path of the system's Program Files directory. $\{pf\}$ is equivalent to $\{pf32\}$ unless the install is running in <u>64-bit</u> <u>mode</u>, in which case it is equivalent to $\{pf64\}$.

{pf32}

32-bit Program Files. The path of the system's 32-bit Program Files directory, typically "C:\Program Files" on 32-bit Windows and "C:\Program Files (x86)" on 64-bit Windows.

{pf64}

64-bit Windows only: 64-bit Program Files. The path of the system's 64-bit Program Files directory, typically "C:\Program Files". An exception will be raised if an attempt is made to expand this constant on 32-bit Windows.

{cf}

Common Files. The path of the system's Common Files directory. $\{cf\}$ is equivalent to $\{cf32\}$ unless the install is running in <u>64-bit</u> <u>mode</u>, in which case it is equivalent to $\{cf64\}$.

{cf32}

32-bit Common Files. The path of the system's 32-bit Common Files directory, typically "C:\Program Files\Common Files" on 32-bit Windows and "C:\Program Files (x86)\Common Files" on 64-bit Windows.

{cf64}

64-bit Windows only: 64-bit Common Files. The path of the system's 64-bit Common Files directory, typically "C:\Program Files\Common Files". An exception will be raised if an attempt is made to expand this constant on 32-bit Windows.

{tmp}

Temporary directory used by Setup or Uninstall. This is *not* the value of the user's TEMP environment variable. It is a subdirectory of the user's temporary directory which is created by Setup or Uninstall at startup (with a name like "C:\WINDOWS\TEMP\ISxxxxx.tmp"). All files and subdirectories in this directory are deleted when Setup or Uninstall exits. During Setup, this is primarily useful for extracting files that are to be executed in the [Run] section but aren't needed after the installation.

{fonts}

Fonts directory. Normally named "FONTS" under the Windows directory.

{dao}

DAO directory. This is equivalent to {cf}\Microsoft Shared\DAO.

{dotnet11}

32-bit .NET Framework version 1.1 root directory.

An exception will be raised if an attempt is made to expand this constant on a system with no .NET Framework version 1.1 present.

{dotnet20}

.NET Framework version 2.0 root directory. {dotnet20} is equivalent to {dotnet2032} unless the install is running in <u>64-bit</u> <u>mode</u>, in which case it is equivalent to {dotnet2064}.

An exception will be raised if an attempt is made to expand this constant on a system with no .NET Framework version 2.0 present.

{dotnet2032}

32-bit .NET Framework version 2.0 root directory.

An exception will be raised if an attempt is made to expand this constant on a system with no .NET Framework version 2.0 present.

{dotnet2064}

64-bit Windows only: 64-bit .NET Framework version 2.0 root directory.

An exception will be raised if an attempt is made to expand this constant on a system with no .NET Framework version 2.0 present.

{dotnet40}

.NET Framework version 4.0 root directory. {dotnet40} is equivalent to {dotnet4032} unless the install is running in <u>64-bit</u> <u>mode</u>, in which case it is equivalent to {dotnet4064}.

An exception will be raised if an attempt is made to expand this constant on a system with no .NET Framework version 4.0 present.

{dotnet4032}

32-bit .NET Framework version 4.0 root directory.

An exception will be raised if an attempt is made to expand this constant on a system with no .NET Framework version 4.0 present.

{dotnet4064}

64-bit Windows only: 64-bit .NET Framework version 2.0 root directory.

An exception will be raised if an attempt is made to expand this constant on a system with no .NET Framework version 4.0 present.

Shell Folder Constants

Inno Setup supports another set of directory constants, referred to as *shell folder constants*. They can be used in the same way as the other directory constants.

"common" below constants refer to the All Users profile.

The "user" constants refer to the profile of the user running Setup. This user is often not the same as the currently logged-in user, so use the "user" constants with caution.

Except where otherwise noted, shell folder constants work on all versions of Windows that Inno Setup supports.

* = The "common" form of this constant is mapped to the "user" form if the logged-in user lacks administrative privileges, or if PrivilegesRequired is set to lowest.

{group}

The path to the Start Menu folder, as selected by the user on Setup's *Select Start Menu Folder* wizard page. This folder is created under the *All Users* profile unless the user installing the application does not have administrative privileges, in which case it is created in the user's profile.

{localappdata}

The path to the local (nonroaming) Application Data folder.

{sendto}

The path to the current user's Send To folder. (There is no common Send To folder.)

{userappdata} & {commonappdata}

The path to the Application Data folder.

{usercf}

The path to the current user's Common Files directory. Only Windows 7 and later supports {usercf}; if used on previous Windows versions, it will translate to the same directory as {localappdata}\Programs\Common.

{userdesktop} & {commondesktop} *

The path to the desktop folder.

{userdocs} & **{commondocs}** The path to the My Documents folder.

{userfavorites} & {commonfavorites} *

The path to the Favorites folder.

{userpf}

The path to the current user's Program Files directory. Only Windows 7 and later supports {userpf}; if used on previous Windows versions, it will translate to the same directory as {localappdata}\Programs.

{userprograms} & {commonprograms} *

The path to the Programs folder on the Start Menu.

{userstartmenu} & {commonstartmenu} *

The path to the top level of the Start Menu.

{userstartup} & {commonstartup} *

The path to the Startup folder on the Start Menu.

{usertemplates} & {commontemplates} *

The path to the Templates folder.

Other Constants

{\}

A backslash character. See the note at the top of this page for an explanation of what the difference between using $\{\\}$ and only a $\$ is.

{%NAME|DefaultValue}

Embeds the value of an environment variable.

- *NAME* specifies the name of the environment variable to use.
- *DefaultValue* determines the string to embed if the specified variable does not exist on the user's system.
- If you wish to include a comma, vertical bar ("|"), or closing brace ("}") inside the constant, you must escape it via "%encoding." Replace the character with a "%" character, followed by its two-digit hex code. A comma is "%2c", a vertical bar is "%7c", and a closing brace is "%7d". If you want to include an actual "%" character, use "%25".
- *NAME* and *DefaultValue* may include constants. Note that you do *not* need to escape the closing brace of a constant as described above; that is only necessary when the closing brace is used elsewhere.

```
Examples:
{%COMSPEC}
{%PROMPT|$P$G}
```

{cmd}

The full pathname of the system's standard command interpreter, *Windows\System32*\cmd.exe. Note that the COMSPEC environment variable is not used when expanding this constant.

{computername}

The name of the computer the Setup or Uninstall program is running on (as returned by the Windows *GetComputerName* function).

{drive:Path}

Extracts and returns the drive letter and colon (e.g. "C:") from the specified path. In the case of a UNC path, it returns the server and share name (e.g. "\\SERVER\SHARE").

- Path specifies the path.
- If you wish to include a comma, vertical bar ("|"), or closing brace ("}") inside the constant, you must escape it via "%-encoding." Replace the character with a "%" character, followed by its two-digit hex code. A comma is "%2c", a vertical bar is "%7c", and a closing brace is "%7d". If you want to include an actual "%" character, use "%25".
- *Path* may include constants. Note that you do *not* need to escape the closing brace of a constant as described above; that is only necessary when the closing brace is used elsewhere.

```
Examples:
```

```
{drive:{src}}
{drive:c:\path\file}
{drive:\\server\share\path\file}
```

{groupname}

The name of the folder the user selected on Setup's *Select Start Menu Folder* wizard page. This differs from {group} in that it is only the name; it does not include a path.

{hwnd}

(Special-purpose) Translates to the window handle of the Setup program's background window.

{wizardhwnd}

(Special-purpose) Translates to the window handle of the Setup wizard window. This handle is set to '0' if the window handle isn't available at the time the translation is done.

{ini:Filename,Section,Key|DefaultValue}

Embeds a value from an .INI file.

- Filename specifies the name of the .INI file to read from.
- Section specifies the name of the section to read from.
- *Key* specifies the name of the key to read.
- *DefaultValue* determines the string to embed if the specified key does not exist.
- If you wish to include a comma, vertical bar ("|"), or closing brace ("}") inside the constant, you must escape it via "%-encoding." Replace the character with a "%" character, followed by its two-digit hex code. A comma is "%2c", a vertical bar is "%7c", and a closing brace is "%7d". If you want to include an actual "%" character, use "%25".
- *Filename, Section,* and *Key* may include constants. Note that you do *not* need to escape the closing brace of a constant as described above; that is only necessary when the closing brace is used elsewhere.

```
Example:
```

{ini:{win}\MyProg.ini,Settings,Path|{pf}\My Progr

{language}

The internal name of the selected language. See the [Languages] section documentation for more information.

{cm:*MessageName*}

{cm:MessageName,Arguments}

Embeds a custom message value based on the active language.

- *MessageName* specifies the name of custom message to read from. See the [CustomMessages] section documentation for more information.
- *Arguments* optionally specifies a comma separated list of arguments to the message value.
- If you wish to include a comma, vertical bar ("|"), or closing brace ("}") inside the constant, you must escape it via "%-

encoding." Replace the character with a "%" character, followed by its two-digit hex code. A comma is "%2c", a vertical bar is "%7c", and a closing brace is "%7d". If you want to include an actual "%" character, use "%25".

• Each argument in *Arguments* may include constants. Note that you do *not* need to escape the closing brace of a constant as described above; that is only necessary when the closing brace is used elsewhere.

Example:

{cm:LaunchProgram,Inno Setup}

The example above translates to "Launch Inno Setup" if English is the active language.

{reg:HKxx\SubkeyName,ValueName|DefaultValue}

Embeds a registry value.

- HKxx specifies the root key; see the [Registry] section documentation for a list of possible root keys.
- SubkeyName specifies the name of the subkey to read from.
- *ValueName* specifies the name of the value to read; leave *ValueName* blank if you wish to read the "default" value of a key.
- DefaultValue determines the string to embed if the specified registry value does not exist, or is not a string type (REG_SZ or REG_EXPAND_SZ).
- If you wish to include a comma, vertical bar ("|"), or closing brace ("}") inside the constant, you must escape it via "%-encoding." Replace the character with a "%" character, followed by its two-digit hex code. A comma is "%2c", a vertical bar is "%7c", and a closing brace is "%7d". If you want to include an actual "%" character, use "%25".
- *SubkeyName, ValueName,* and *DefaultValue* may include constants. Note that you do *not* need to escape the closing

brace of a constant as described above; that is only necessary when the closing brace is used elsewhere.

```
Example:
{reg:HKLM\Software\My Program,Path|{pf}\My Progra
```

{param:ParamName|DefaultValue}

Embeds a command line parameter value.

- *ParamName* specifies the name of the command line parameter to read from.
- *DefaultValue* determines the string to embed if the specified command line parameter does not exist, or its value could not be determined.
- If you wish to include a comma, vertical bar ("|"), or closing brace ("}") inside the constant, you must escape it via "%-encoding." Replace the character with a "%" character, followed by its two-digit hex code. A comma is "%2c", a vertical bar is "%7c", and a closing brace is "%7d". If you want to include an actual "%" character, use "%25".
- *ParamName* and *DefaultValue* may include constants. Note that you do *not* need to escape the closing brace of a constant as described above; that is only necessary when the closing brace is used elsewhere.

```
Example:
```

{param:Path|{pf}\My Program}

The example above translates to $c:\My$ Program if the command line /Path="c:\My Program" was specified.

{srcexe}

The full pathname of the Setup program file, e.g. "C:\SETUP.EXE".

{uninstallexe}

The full pathname of the uninstall program extracted by Setup, e.g. "C:\Program Files\My Program\unins000.exe". This constant is

typically used in an [Icons] section entry for creating an Uninstall icon. It is only valid if Uninstallable is yes (the default setting).

{sysuserinfoname} {sysuserinfoorg}

The name and organization, respectively, that Windows is registered to. This information is read from the registry.

{userinfoname} {userinfoorg} {userinfoserial}

The name, organization and serial number, respectively, that the user entered on the *User Information* wizard page (which can be enabled via the UserInfoPage directive). Typically, these constants are used in [Registry] or [INI] entries to save their values for later use.

{username}

The name of the user who is running Setup or Uninstall program (as returned by the *GetUserName* function).

{log}

The log file name, or an empty string if <u>logging</u> is not enabled.

Common Parameters

There are three optional <u>parameters</u> that are supported by all sections whose entries are separated into parameters. They are:

Languages

A space separated list of language names, telling Setup to which languages the entry belongs. If the end user selects a language from this list, the entry is processed (for example: the file is installed).

An entry without a Languages parameter is always processed, unless other parameters say it shouldn't be.

Besides space separated lists, you may also use boolean expressions. See <u>Components and Tasks parameters</u> for examples of boolean expressions.

Example: Languages: en nl

MinVersion

A minimum <u>Windows version</u> for the entry to be processed. If you use "0" then the entry will never be processed. <u>Build numbers</u> <u>and/or service pack levels</u> may be included. This overrides any MinVersion directive in the script's [Setup] section.

An entry without a MinVersion parameter is always processed, unless other parameters say it shouldn't be.

For compatibility with previous versions of Inno Setup, separate Windows 95/98/Me and Windows NT version numbers may be specified, separated by a comma. Example: 4.1, 5.0. The Windows 95/98/Me version number (the first number) isn't used, however, as Inno Setup no longer supports Windows 95/98/Me.

Example:

MinVersion: 5.1

OnlyBelowVersion

Essentially the opposite of MinVersion. Specifies the minimum <u>Windows version</u> for the entry *not* to be processed. For example, if you put 6.0 and the user is running Windows 2000 or XP, the entry *will* be processed, but if the user is running Windows Vista (which reports its version as 6.0) or later, it will *not* be processed. Putting "0" means there is no upper version limit. <u>Build numbers</u> and/or service pack levels may be included. This overrides any OnlyBelowVersion directive in the script's [Setup] section.

An entry without an OnlyBelowVersion parameter is always processed, unless other parameters say it shouldn't be.

For compatibility with previous versions of Inno Setup, separate Windows 95/98/Me and Windows NT version numbers may be specified, separated by a comma. Example: 4.1,5.0. The Windows 95/98/Me version number (the first number) isn't used, however, as Inno Setup no longer supports Windows 95/98/Me.

Example:

OnlyBelowVersion: 6.0

Components and Tasks Parameters

There are two optional <u>parameters</u> that are supported by all sections whose entries are separated into parameters, except [Types], [Components] and [Tasks]. They are:

Components

A space separated list of component names, telling Setup to which components the entry belongs. If the end user selects a component from this list, the entry is processed (for example: the file is installed).

An entry without a Component's parameter is always processed, unless other parameters say it shouldn't be.

```
Example:
[Files]
Source: "MYPROG.EXE"; DestDir: "{app}"; Component
Source: "MYPROG.CHM"; DestDir: "{app}"; Component
Source: "README.TXT"; DestDir: "{app}"
```

Tasks

A space separated list of task names, telling Setup to which task the entry belongs. If the end user selects a task from this list, the entry is processed (for example: the file is installed).

An entry without a Tasks parameter is always processed, unless other parameters say it shouldn't be.

Note that the *Don't create a Start Menu folder* checkbox on the *Select Start Menu Folder* wizard page doesn't affect [Icons] entries that have Tasks parameters since they have their own checkboxes.

```
Example:
```

```
[Icons]
Name: "{group}\My Program"; Filename: "{app}\MyPr
```

Besides space separated lists, you may also use boolean expressions as Components and Tasks parameters. Supported operators include not, and, and or. For example:

```
[Components]
Name: a; Description: a
Name: b; Description: b
[Tasks]
Name: p; Description: a or b; Components: a or b
Name: q; Description: a and b; Components: a and
Name: r; Description: not a or b; Components: not
Name: s; Description: not (a or b); Components: n
Name: t; Description: a or b - old style; Components: n
```

[Setup] section

This section contains global settings used by the installer and uninstaller. Certain directives are required for any installation you create. Here is an example of a [Setup] section:

```
[Setup]
AppName=My Program
AppVersion=1.5
DefaultDirName={pf}\My Program
DefaultGroupName=My Program
```

By default, any leading or trailing whitespace in a directive's value will be stripped. It is possible to avoid this by surrounding the directive's value in double quotes (").

The following directives can be placed in the [Setup] section:

(**bold** = required)

Compiler-related

- <u>Compression</u>
- <u>CompressionThreads</u>
- DiskClusterSize
- DiskSliceSize
- DiskSpanning
- Encryption
- InternalCompressLevel
- LZMAAlgorithm
- LZMABlockSize
- LZMADictionarySize
- LZMAMatchFinder
- <u>LZMANumBlockThreads</u>
- LZMANumFastBytes
- <u>LZMAUseSeparateProcess</u>
- <u>MergeDuplicateFiles</u>
- Output
- OutputBaseFilename
- <u>OutputDir</u>
- OutputManifestFile
- <u>ReserveBytes</u>
- <u>SignedUninstaller</u>
- <u>SignedUninstallerDir</u>
- <u>SignTool</u>
- <u>SignToolRetryCount</u>
- <u>SlicesPerDisk</u>
- SolidCompression
- <u>SourceDir</u>
- <u>TerminalServicesAware</u>
- <u>UseSetupLdr</u>
- <u>VersionInfoCompany</u>
- VersionInfoCopyright
- <u>VersionInfoDescription</u>

- <u>VersionInfoProductName</u>
- <u>VersionInfoProductTextVersion</u>
- <u>VersionInfoProductVersion</u>
- <u>VersionInfoTextVersion</u>
- <u>VersionInfoVersion</u>

Installer-related

Functional: These directives affect the operation of the Setup program, or are saved and used later by the uninstaller.

- <u>AllowCancelDuringInstall</u>
- <u>AllowNetworkDrive</u>
- <u>AllowNolcons</u>
- <u>AllowRootDirectory</u>
- <u>AllowUNCPath</u>
- <u>AlwaysRestart</u>
- <u>AlwaysShowComponentsList</u>
- <u>AlwaysShowDirOnReadyPage</u>
- <u>AlwaysShowGroupOnReadyPage</u>
- <u>AlwaysUsePersonalGroup</u>
- <u>AppendDefaultDirName</u>
- <u>AppendDefaultGroupName</u>
- <u>AppComments</u>
- <u>AppContact</u>
- Appld
- AppModifyPath
- <u>AppMutex</u>
- <u>AppName</u>
- <u>AppPublisher</u>
- <u>AppPublisherURL</u>
- <u>AppReadmeFile</u>
- <u>AppSupportPhone</u>
- <u>AppSupportURL</u>
- <u>AppUpdatesURL</u>
- <u>AppVerName</u>
- <u>AppVersion</u>
- <u>ArchitecturesAllowed</u>
- <u>ArchitecturesInstallIn64BitMode</u>
- <u>ChangesAssociations</u>
- <u>ChangesEnvironment</u>

- <u>CloseApplications</u>
- **CloseApplicationsFilter**
- <u>CreateAppDir</u>
- <u>CreateUninstallRegKey</u>
- DefaultDialogFontName
- DefaultDirName
- DefaultGroupName
- DefaultUserInfoName
- DefaultUserInfoOrg
- DefaultUserInfoSerial
- DirExistsWarning
- DisableDirPage
- <u>DisableFinishedPage</u>
- <u>DisableProgramGroupPage</u>
- <u>DisableReadyMemo</u>
- <u>DisableReadyPage</u>
- DisableStartupPrompt
- DisableWelcomePage
- EnableDirDoesntExistWarning
- ExtraDiskSpaceRequired
- InfoAfterFile
- InfoBeforeFile
- LanguageDetectionMethod
- LicenseFile
- <u>MinVersion</u>
- OnlyBelowVersion
- Password
- <u>PrivilegesRequired</u>
- <u>RestartApplications</u>
- <u>RestartIfNeededByRun</u>
- <u>SetupLogging</u>
- <u>SetupMutex</u>
- <u>ShowLanguageDialog</u>
- <u>ShowUndisplayableLanguages</u>

- <u>TimeStampRounding</u>
- <u>TimeStampsInUTC</u>
- <u>TouchDate</u>
- <u>TouchTime</u>
- Uninstallable
- <u>UninstallDisplayIcon</u>
- <u>UninstallDisplayName</u>
- <u>UninstallDisplaySize</u>
- <u>UninstallFilesDir</u>
- <u>UninstallLogMode</u>
- <u>UninstallRestartComputer</u>
- <u>UpdateUninstallLogAppName</u>
- <u>UsePreviousAppDir</u>
- <u>UsePreviousGroup</u>
- <u>UsePreviousLanguage</u>
- <u>UsePreviousSetupType</u>
- <u>UsePreviousTasks</u>
- <u>UsePreviousUserInfo</u>
- <u>UserInfoPage</u>

Cosmetic: These directives only affect the appearance of the Setup program.

- AppCopyright
- BackColor
- BackColor2
- <u>BackColorDirection</u>
- BackSolid
- FlatComponentsList
- <u>SetupIconFile</u>
- <u>ShowComponentSizes</u>
- <u>ShowTasksTreeLines</u>
- <u>WindowShowCaption</u>
- <u>WindowStartMaximized</u>
- <u>WindowResizable</u>
- <u>WindowVisible</u>

- <u>WizardImageBackColor</u>
- <u>WizardImageFile</u>
- <u>WizardImageStretch</u>
- <u>WizardSmallImageFile</u>

Obsolete

These directives are obsolete and should not be used in any new scripts.

- <u>AlwaysCreateUninstallIcon</u>
- <u>DisableAppendDir</u>
- DontMergeDuplicateFiles
- <u>MessagesFile</u>
- UninstallIconFile
- <u>UninstallIconName</u>
- UninstallStyle
- <u>WizardSmallImageBackColor</u>
- <u>WizardStyle</u>

[Setup]: AllowCancelDuringInstall

Valid values: yes or no

Default value: yes

Description:

Setting this to no prevents the user from cancelling during the actual installation process, by disabling the Cancel button and ignoring clicks on the close button. This has the same effect as passing /NOCANCEL to Setup on the command line.

[Setup]: AllowNetworkDrive

Valid values: <u>yes or no</u>

Default value: yes

Description:

If set to no, the user will not be allowed to enter a network drive on the *Select Destination Location* page of the wizard.

To fully disallow installation to network locations, you must also set <u>AllowUNCPath</u> to no.

[Setup]: AllowNolcons

Valid values: <u>yes or no</u>

Default value: no

Description:

When set to yes, Setup will display a *Don't create a Start Menu folder* check box on the *Select Start Menu Folder* wizard page, which allows the user to skip creation of program shortcuts on the Start Menu.

Only [Icons] entries that have a Name parameter starting with {group}\ and no Tasks parameter are affected by default. To force the check box to have an effect on a particular [Icons] entry, add a Check: not <u>WizardNoIcons</u> parameter.

[Setup]: AllowRootDirectory

Valid values: yes or no

Default value: no

Description:

When set to no, the default, the user will not be allowed to enter a root directory (such as "C:\") on the *Select Destination Location* page of the wizard.
[Setup]: AllowUNCPath

Valid values: <u>yes or no</u>

Default value: yes

Description:

If set to no, the user will not be allowed to enter a UNC path (such as "\\server\share") on the *Select Destination Location* page of the wizard. This was the default behavior in Inno Setup 2.0.17 and earlier.

To fully disallow installation to network locations, you must also set <u>AllowNetworkDrive</u> to no.

[Setup]: AlwaysCreateUninstallIcon

Description:

Obsolete in 3.0. This directive is no longer supported. If you wish to create an Uninstall icon, use the new {uninstallexe} constant in the Filename parameter of an [lcons] section entry.

[Setup]: AlwaysRestart

Valid values: <u>yes or no</u>

Default value: no

Description:

When set to yes, Setup will always prompt the user to restart the system at the end of a successful installation, regardless of whether this is necessary (for example, because of [Files] section entries with the restartreplace flag).

[Setup]: AlwaysShowComponentsList

Valid values: yes or no

Default value: yes

Description:

If this directive is set to yes, Setup will always show the components list for customizable setups. If this is set to no Setup will only show the components list if the user selected a custom type from the type list.

[Setup]: AlwaysShowDirOnReadyPage

Valid values: yes or no

Default value: no

Description:

If this directive is set to yes, Setup will always show the selected directory in the list of settings on the *Ready to Install* wizard page. If this is set to no, Setup will not show the selected directory if DisableDirPage is yes.

[Setup]: AlwaysShowGroupOnReadyPage

Valid values: <u>yes or no</u>

Default value: no

Description:

If this directive is set to yes, Setup will always show the selected Start Menu folder name in the list of settings on the *Ready to Install* wizard page. If this is set to no, Setup will not show the selected Start Menu folder name if DisableProgramGroupPage is yes.

If no Start Menu folder is going to be created by Setup, this directive is effectively ignored.

[Setup]: AlwaysUsePersonalGroup

Valid values: yes or no

Default value: no

Description:

Normally, Inno Setup's {group} constant points to the All Users start menu if the user has administrative privileges. If this directive is set to yes, it always uses current user's profile.

[Setup]: AppComments

Description:

This string is displayed on the "Support" dialog of the *Add/Remove Programs* Control Panel applet. The value may include constants.

Example:

AppComments=Hello.

[Setup]: AppContact

Description:

This string is displayed on the "Support" dialog of the *Add/Remove Programs* Control Panel applet. The value may include constants.

Example:

AppContact=My Company Customer Support

[Setup]: AppCopyright

Description:

Specifies a copyright message that Setup will display in the bottomright corner of Setup's background window when <u>WindowVisible</u> is yes.

The value of this directive is also used as the default value for the <u>VersionInfoCopyright</u> directive if it is not specified.

Example:

```
AppCopyright=Copyright (C) 1997-2005 My Company, Inc
```

[Setup]: AppendDefaultDirName

Valid values: yes or no

Default value: yes

Description:

By default, when a folder in the dialog displayed by the *Browse…* button on the *Select Destination Location* wizard page is clicked, Setup automatically appends the last component of DefaultDirName onto the new path. For example, if DefaultDirName is {pf}\My Program and "Z:\" is clicked, the new path will become "Z:\My Program".

Setting this directive to no disables the aforementioned behavior. In addition, it causes a *Make New Folder* button to appear on the dialog.

[Setup]: AppendDefaultGroupName

Valid values: yes or no

Default value: yes

Description:

By default, when a folder in the dialog displayed by the *Browse…* button on the *Select Start Menu Folder* wizard page is clicked, Setup automatically appends the last component of DefaultGroupName onto the new path. For example, if DefaultGroupName is My Program and "Accessories" is clicked, the new path will become "Accessories\My Program".

Setting this directive to no disables the aforementioned behavior. In addition, it causes a *Make New Folder* button to appear on the dialog.

[Setup]: AppId

Default value: <u>AppName</u>

Description:

The value of AppId is stored inside uninstall log files (unins???.dat), and is checked by subsequent installations to determine whether it may <u>append to a particular existing uninstall log</u>. Setup will only append to an uninstall log if the AppId of the existing uninstall log is the same as the current installation's AppId. For a practical example, say you have two installations -- one entitled *My Program* and the other entitled *My Program 1.1 Update*. To get My Program 1.1 Update to append to My Program's uninstall log, you would have to set AppId to the same value in both installations.

AppId also determines the actual name of the Uninstall registry key, to which Inno Setup tacks on "_is1" at the end. (Therefore, if AppId is "MyProgram", the key will be named "MyProgram_is1".) Pre-1.3 versions of Inno Setup based the key name on the value of AppVerName.

AppId is a not used for display anywhere, so feel free to make it as cryptic as you desire. The value may include constants.

If you use a {code:..} constant to allow your user to customize AppId, you do not need to return the real value until just before the installation starts: if necessary you may return an empty or generic value at earlier times. If not empty, this value will only be used to attempt to restore previous install settings (like the settings stored by [Setup] section directive <u>UsePreviousAppDir</u>). If empty, it isn't used for anything.

The length of AppId with all constants evaluated should never exceed 127 characters.

Example:

AppId=MyProgram

[Setup]: AppModifyPath

Description:

When this directive is set, a separate "Modify" button in the Add/Remove Programs Control Panel applet will be displayed. Setting it is optional. The value may include constants.

Example:

AppModifyPath="{app}\Setup.exe" /modify=1

[Setup]: AppMutex

Description:

This directive is used to prevent the user from installing new versions of an application while the application is still running, and to prevent the user from uninstalling a running application. It specifies the names of one or more named mutexes (multiple mutexes are separated by commas), which Setup and Uninstall will check for at startup. If any exist, Setup/Uninstall will display the message: "[Setup or Uninstall] has detected that [AppName] is currently running. Please close all instances of it now, then click OK to continue, or Cancel to exit." The value may include constants.

Use of this directive requires that you add code to your application which creates a mutex with the name you specify in this directive. Examples of creating a mutex in Delphi, C, and Visual Basic are shown below. The code should be executed during your application's startup.

Delphi:

CreateMutex(nil, False, 'MyProgramsMutexName');

C:

```
CreateMutex(NULL, FALSE, "MyProgramsMutexName");
```

Visual Basic (submitted by Peter Young):

```
'Place in Declarations section:
Private Declare Function CreateMutex Lib "kernel3
Alias "CreateMutexA" _
(ByVal lpMutexAttributes As Long, _
ByVal bInitialOwner As Long, _
ByVal lpName As String) As Long
'Place in startup code (Form_Load or Sub Main):
CreateMutex 0&, 0&, "MyProgramsMutexName"
```

It is not necessary to explicitly destroy the mutex object upon your

application's termination; the system will do this automatically. Nor is it recommended that you do so, because ideally the mutex object should exist until the process completely terminates.

Note that mutex name comparison in Windows is case sensitive.

To specify a mutex name containing a comma, escape the comma with a backslash.

See the topic for CreateMutex in the MS SDK help for more information on mutexes.

Example:

AppMutex=MyProgramsMutexName

See also: <u>SetupMutex</u>

[Setup]: AppName

Description:

This required directive specifies the name of the application being installed. Do not include the version number, as that is defined by the <u>AppVersion</u> and/or <u>AppVerName</u> directives. AppName is displayed throughout the Setup program and uninstaller in window titles, wizard pages, and dialog boxes. The value may include constants.

The value of this directive is also used as the default value for the <u>AppId</u>, <u>VersionInfoDescription</u>, and <u>VersionInfoProductName</u> directives if those are not specified.

Example:

AppName=My Program

[Setup]: AppPublisher

Description:

This string is displayed on the "Support" dialog of the *Add/Remove Programs* Control Panel applet. The value may include constants.

The value of this directive is also used as the default value for the <u>VersionInfoCompany</u> directive if it is not specified.

Example:

AppPublisher=My Company, Inc.
AppPublisherURL=http://www.example.com/

[Setup]: AppPublisherURL

Description:

A link to the specified URL is displayed on the "Support" dialog of the *Add/Remove Programs* Control Panel applet. The value may include constants.

Example:

AppPublisher=My Company, Inc.
AppPublisherURL=http://www.example.com/

[Setup]: AppReadmeFile

Description:

This string, which may be a URL, is displayed on the "Support" dialog of the *Add/Remove Programs* Control Panel applet. The value may include constants.

Example:

AppReadmeFile=http://www.example.com/readme.html

[Setup]: AppSupportPhone

Description:

This string is displayed on the "Support" dialog of the *Add/Remove Programs* Control Panel applet. The value may include constants.

Example:

AppSupportPhone=1-800-555-1212

[Setup]: AppSupportURL

Description:

A link to the specified URL is displayed on the "Support" dialog of the *Add/Remove Programs* Control Panel applet. The value may include constants.

Example:

AppSupportURL=http://www.example.com/support.html

[Setup]: AppUpdatesURL

Description:

A link to the specified URL is displayed on the "Support" dialog of the *Add/Remove Programs* Control Panel applet. The value may include constants.

Example:

AppUpdatesURL=http://www.example.com/updates.html

[Setup]: AppVerName

Default value: AppName version AppVersion, localized according to the active language's NameAndVersion <u>custom</u> message

Description:

This directive specifies the name of the application plus its version number. The value of this directive is displayed on the *Welcome* page of Setup's wizard, and is used as the default title of the application's *Add/Remove Programs* entry (see <u>UninstallDisplayName</u>). The value may include constants.

This directive is required if the <u>AppVersion</u> directive is not set.

```
Examples:
AppVerName=My Program 1.5
AppVerName=My Program version 1.5
AppVerName={cm:NameAndVersion,My Program,1.5}
```

[Setup]: AppVersion

Description:

This directive specifies the version number of the application being installed. The value of this directive, which may include constants, is used in the default value for the <u>AppVerName</u> directive, and is displayed in the Version field of the application's *Add/Remove Programs* entry. It is also used to set the MajorVersion and MinorVersion values in the Uninstall registry key when possible.

This directive is required and cannot be empty if the <u>AppVerName</u> directive is not set.

Example:

AppVersion=1.5

[Setup]: ArchitecturesAllowed

Valid values: One or more of the following, separated by spaces: x86 x64 ia64

Default value: (blank)

Description:

Specifies which processor architecture(s) Setup is allowed to run on. If this directive is not specified or is blank, Setup will be allowed to run on all processor architectures capable of executing its 32-bit code (including ones it doesn't recognize). Otherwise, if a user's processor architecture is not one of those specified in this directive, Setup will display an error message and exit.

If your application's binaries are all 32-bit and run in user mode, you should not change this directive from its default value (specifically, don't set it to x86), because normally such binaries will run without issue on the x64 and Itanium editions of Windows via the WOW64 emulator.

If you install any 32-bit device drivers, you should set this directive to x86, as 32-bit device drivers cannot function on 64-bit Windows.

If your application's binaries are built for the x64 or Itanium architectures, you should set this directive to either x64 or ia64 respectively.

See also:

ArchitecturesInstallIn64BitMode

[Setup]: ArchitecturesInstallIn64BitMode

Valid values: One or more of the following, separated by spaces: x64 ia64

Default value: (blank)

Description:

Specifies the 64-bit processor architecture(s) on which Setup should install in <u>64-bit mode</u>. If this directive is not specified or is blank, Setup will always install in <u>32-bit mode</u>.

Normally, you should not change this directive from its default value unless your application contains native 64-bit binaries.

Be sure you have read the <u>64-bit Installation Limitations</u> topic before setting this directive.

If your application runs only on 64-bit processor architectures, you should set <u>ArchitecturesAllowed</u> to the same value as this directive to prevent Setup from running on 32-bit Windows.

Setup can only run in 64-bit mode on versions of Windows that provide the API support Inno Setup requires (e.g. RegDeleteKeyEx). All x64 editions provide the necessary APIs, however Itanium editions prior to Windows Server 2003 SP1 do not. If the user is running an older Itanium version of Windows, Setup will display a message (MissingW0W64APIs) recommending that the user install a service pack, and exit.

See also:

ArchitecturesAllowed

[Setup]: BackColor, BackColor2

- Valid values: A value in the form of \$bbggrr, where rr, gg, and bb specify the two-digit intensities (in hexadecimal) for red, green, and blue respectively. Or it may be one of the following predefined color names: clBlack, clMaroon, clGreen, clOlive, clNavy, clPurple, clTeal, clGray, clSilver, clRed, clLime, clYellow, clBlue, clFuchsia, clAqua, clWhite.
- Default value: clBlue for BackColor, clBlack for BackColor2

Description:

The BackColor directive specifies the color to use at the top (or left, if BackColorDirection=lefttoright) of the setup window's gradient background. BackColor2 specifies the color to use at the bottom (or right).

The setting of BackColor2 is ignored if BackSolid=yes.

Examples: BackColor=clBlue BackColor2=clBlack

BackColor=\$FF0000 BackColor2=\$000000

[Setup]: BackColorDirection

Valid values: toptobottom or lefttoright

Default value: toptobottom

Description:

This determines the direction of the gradient background on the setup window. If BackColorDirection is toptobottom, it is drawn from top to bottom; if it is lefttoright, it is drawn from left to right.

[Setup]: BackSolid

Valid values: <u>yes or no</u>

Default value: no

Description:

This specifies whether to use a solid or gradient background on the setup window. If this is yes, the background is a solid color (the color specified by BackColor; BackColor2 is ignored).

[Setup]: ChangesAssociations

Valid values: yes or no

Default value: no

Description:

When set to yes, Setup will tell Explorer to refresh its file associations information at the end of the installation, and Uninstall will do the same at the end of uninstallation.

If your installation creates a file association but doesn't have ChangesAssociations set to yes, the correct icon for the file type likely won't be displayed until the user logs off or restarts the computer.

[Setup]: ChangesEnvironment

Valid values: yes or no

Default value: no

Description:

When set to yes, at the end of the installation Setup will notify other running applications (notably Windows Explorer) that they should reload their environment variables from the registry.

If your installation creates or changes an environment variable but doesn't have ChangesEnvironment set to yes, the new/changed environment variable will not be seen by applications launched from Explorer until the user logs off or restarts the computer.

[Setup]: CloseApplications

Valid values: <u>yes or no</u>

Default value: yes

Description:

If set to yes and Setup is not running silently, Setup will pause on the *Preparing to Install* wizard page if it detects applications using files that need to be updated by the [Files] or [InstallDelete] section, showing the applications and asking the user if Setup should automatically close the applications and restart them after the installation has completed.

If set to yes and Setup is running silently, Setup will always close and restart such applications, unless told not to via the command line.

Note: Setup uses the Windows <u>Restart Manager</u> API to close and restart applications, which is available only on Windows Vista and newer.

See also: <u>CloseApplicationsFilter</u> <u>RegisterExtraCloseApplicationsResources</u> <u>RestartApplications</u>

[Setup]: CloseApplicationsFilter

Valid values: A list of file name wildcards, separated by commas

Default value: *.exe, *.dll, *.chm

Description:

Controls which files Setup will check for being in use.

Setting this to *. * can provide better checking at the expense of speed.

See also: <u>CloseApplications</u> <u>RestartApplications</u>

[Setup]: Compression

Valid values:	zip zip/1 through zip/9	
vana vanacs.	bzip	
	bzip/1 through bzip/9	
	lzma	
	lzma/fast	
	lzma/normal	
	lzma/max	
	lzma/ultra (review memory requirements below	
	before using)	
	lzma/ultra64 (review memory requirements below	
	before using)	
	lzma2	
	lzma2/fast	
	lzma2/normal	
	lzma2/max	
	lzma2/ultra (review memory requirements below	
	before using)	
	1zma2/ultra64 (review memory requirements	
	below before using)	
	none	

Default value: lzma2/max

Description:

This specifies the method of compression to use on the files, and optionally the level of compression. Higher levels compress better but take longer doing so, and may also require more memory while compressing/decompressing.

zip is the method of compression employed by .zip files ("deflate"). It is fast in both compression and decompression, and has very low memory requirements (less than 1 MB for both compression and decompression at level 9), but generally does not compress nearly as well as the other supported methods. zip, like 1zma2, has one special property, though: it will not expand uncompressible data (e.g., files that are already compressed). If a compression level isn't specified, it defaults to 7.

bzip is the method of compression employed by the <u>bzip2</u> compressor. It almost always compresses better than zip but is usually slower in both compression and decompression. Up to 4 MB of memory is required during decompression, and up to 8 MB during compression. If a compression level isn't specified, it defaults to 9.

1zma is the method of compression employed by the 7-Zip LZMA a compressor. It typically compresses significantly better than the zip and bzip methods. However, depending on the compression level used, it can be significantly slower at compressing, and consume a *lot* more memory. The following table summarizes the approximate memory requirements for each of the supported 1zma compression levels. If a compression level isn't specified, it defaults to max.

	Decompression (dictionary	<u>Compression</u>
	<u>size)</u>	
fast (worst)	32 KB	3 MB
normal	2 MB	26 MB
max (default)	8 MB	95 MB
ultra	32 MB	372 MB
ultra64 (best)	64 MB	676 MB

1zma2 is the method of compression employed by the <u>7-Zip LZMA2</u> compressor. LZMA2 is a modified version of LZMA that offers a better compression ratio for uncompressible data (random data expands about 0.005%, compared to 1.35% with original LZMA), and optionally can compress multiple parts of large files in parallel, greatly increasing compression speed but with a possible reduction in compression ratio (see <u>LZMANumBlockThreads</u>). Like LZMA, it can consume a *lot* of memory; see the above table. If a compression level isn't specified, it defaults to max.

none specifies that no compression be used.

See also: SolidCompression LZMAAlgorithm
LZMABlockSize LZMADictionarySize LZMAMatchFinder LZMANumBlockThreads LZMANumFastBytes LZMAUseSeparateProcess

[Setup]: CompressionThreads

Valid values: auto 1 2 (or higher)

Default value: auto

Description:

Controls whether the multi-threaded match finder is enabled on the LZMA and LZMA2 compressors. Enabling the multi-threaded match finder can speed up the compression process by 50% or more on systems with multiple processor cores, and 20% or more on systems with Intel processors featuring Hyper-Threading Technology.

A value of auto (the default) enables the multi-threaded match finder for all compression levels except fast, which doesn't support it.

A value of 1 always disables the multi-threaded match finder.

Values of 2 or higher are currently equivalent to auto.

Note that for the LZMA2 compressor, this directive only controls whether the multi-threaded match finder is used. To enable support for compressing multiple parts of large files in parallel, set <u>LZMANumBlockThreads</u>.

See also:

LZMANumBlockThreads

[Setup]: CreateAppDir

Valid values: <u>yes or no</u>

Default value: yes

Description:

If this is set to no, no directory for the application will be created, the *Select Destination Location* wizard page will not be displayed, and the {app} directory constant is equivalent to the {win} directory constant. If the uninstall feature is enabled when CreateAppDir is no, the uninstall data files are created in the system's Windows directory.

[Setup]: CreateUninstallRegKey

Valid values: <u>yes or no</u>, or a <u>boolean expression</u>

Default value: yes

Description:

If this is set to no or to a <u>boolean expression</u> evaluating to False, Setup won't create an entry in the *Add/Remove Programs* Control Panel applet.

Setting this to no can be useful if your installation is merely an update to an existing application and you don't want another entry created, but don't want to the disable the uninstall features entirely (via Uninstallable=no). In this case, <u>UpdateUninstallLogAppName</u> is usually set to no as well.

See also: Uninstallable

[Setup]: DefaultDialogFontName

Default value: Tahoma

Description:

Specifies the name of the font that should be used in dialogs on languages that do not set DialogFontName in their [LangOptions] <u>section</u>.

If the specified font name does not exist on the user's system or is an empty string, 8-point Microsoft Sans Serif or MS Sans Serif will be substituted.

Prior to the introduction of the DefaultDialogFontName directive in Inno Setup 5.3.9, the default dialog font name was always an empty string.

[Setup]: DefaultDirName

Description:

The value of this required directive is used for the default directory name, which is used in the *Select Destination Location* page of the wizard. Normally it is prefixed by a directory constant.

If <u>UsePreviousAppDir</u> is yes (the default) and Setup finds a previous version of the <u>same application</u> is already installed, it will substitute the default directory name with the directory selected previously.

Examples:

If you used: DefaultDirName={sd}\MYPROG

In Setup, this would typically display: C:\MYPROG

If you used: DefaultDirName={pf}\My Program

In Setup, this would typically display:

C:\Program Files\My Program

[Setup]: DefaultGroupName

Description:

The value of this directive is used for the default Start Menu folder name on the *Select Start Menu Folder* page of the wizard. If this directive is blank or isn't specified, it will use "(Default)" for the name.

Keep in mind that Start Menu folders are stored as literal directories so any characters not allowed in normal directory names can't be used in Start Menu folder names.

Example:

DefaultGroupName=My Program

[Setup]: DefaultUserInfoName

Default value: {sysuserinfoname}

Description:

Specifies the default name shown on the User Information wizard page. This can include constants.

[Setup]: DefaultUserInfoOrg

Default value: {sysuserinfoorg}

Description:

Specifies the default organization shown on the User Information wizard page. This can include constants.

[Setup]: DefaultUserInfoSerial

Description:

Specifies the default serial number shown on the User Information wizard page. This can include constants.

[Setup]: DirExistsWarning

Valid values: auto, <u>yes, or no</u>

Default value: auto

Description:

When set to auto, the default setting, Setup will show a "The directory ... already exists. Would you like to install to that directory anyway?" message if the user selects a directory that already exists on the *Select Destination Location* wizard page, except when another version of the <u>same application</u> is already installed and the selected directory is the same as the previous one (only if UsePreviousAppDir is yes, the default setting).

When set to yes, Setup will always display the "Directory Exists" message when the user selects an existing directory.

When set to no, Setup will never display the "Directory Exists" message.

[Setup]: DisableAppendDir

Valid values: <u>yes or no</u>

Default value: no

Description:

Obsolete in 4.1.2. Pre-4.1.2 versions of Inno Setup had a different directory selection interface, and the DisableAppendDir directive was used to control its behaviour.

[Setup]: DisableDirPage

Valid values: auto, <u>yes, or no</u>

Default value: no

Description:

If this is set to yes, Setup will not show the Select Destination Location wizard page.

If this is set to auto, at startup Setup will look in the registry to see if the <u>same application</u> is already installed, and if so, it will not show the *Select Destination Location* wizard page.

If the *Select Destination Location* wizard page is not shown, it will always use the default directory name.

Also see <u>AlwaysShowDirOnReadyPage</u>.

[Setup]: DisableFinishedPage

Valid values: <u>yes or no</u>

Default value: no

Description:

If this is set to yes, Setup will not show the *Setup Completed* wizard page, and instead will immediately close the Setup program once the installation process finishes. This may be useful if you execute a program in the [Run] section using the nowait flag, and don't want the *Setup Completed* window to remain in the background after the other program has started.

Note that the DisableFinishedPage directive is ignored if a restart of the computer is deemed necessary, or if a file is assigned to the InfoAfterFile [Setup] section directive. In those cases, the Setup Completed wizard page will still be displayed.

[Setup]: DisableProgramGroupPage

Valid values: auto, <u>yes, or no</u>

Default value: no

Description:

If this is set to yes, Setup will not show the *Select Start Menu Folder* wizard page.

If this is set to auto, at startup Setup will look in the registry to see if the <u>same application</u> is already installed, and if so, it will not show the *Select Start Menu Folder* wizard page.

If the *Select Start Menu Folder* wizard page is not shown, it will always use the default Start Menu folder name.

Also see <u>AlwaysShowGroupOnReadyPage</u>.

[Setup]: DisableReadyMemo

Valid values: <u>yes or no</u>

Default value: no

Description:

If this is set to yes, Setup will not show a list of settings on the *Ready to Install* wizard page. Otherwise the list is shown and contains information like the chosen setup type and the chosen components.

[Setup]: DisableReadyPage

Valid values: <u>yes or no</u>

Default value: no

Description:

If this is set to yes, Setup will not show the *Ready to Install* wizard page.

When Setup is not running silently, this directive is ignored if no other wizard page before the *Ready to Install* wizard page has been shown yet.

Setting this to yes does not automatically change the caption of the *Next* button on the new last pre-installation wizard page to *Install*. You must do so manually instead. For example, if the new last pre-installation wizard page is the *Select Program Group* page:

Example:

```
[Setup]
DisableReadyPage=yes
[Code]
procedure CurPageChanged(CurPageID: Integer);
begin
  if CurPageID = wpSelectProgramGroup then
    WizardForm.NextButton.Caption := SetupMessage(ms(
    else
    WizardForm.NextButton.Caption := SetupMessage(ms(
    end;
*
```

[Setup]: DisableStartupPrompt

Valid values: yes or no

Default value: yes

Description:

When this is set to yes, Setup will not show the *This will install...* Do you wish to continue? prompt.

This setting has no effect if UseSetupLdr is set to no.

[Setup]: DisableWelcomePage

Valid values: yes or no

Default value: no

Description:

If this is set to yes, Setup will not show the *Welcome* wizard page.

[Setup]: DiskClusterSize

Default value: 512 (the standard cluster size for floppy disks)

Description:

This specifies the cluster size of the disk media. The Setup Compiler needs to know this in order to properly fill each disk to capacity.

This directive is ignored if disk spanning is not enabled using the DiskSpanning [Setup] section directive.

[Setup]: DiskSliceSize

Valid values: 262144 through 2100000000, or max

Default value: max (210000000)

Description:

This specifies the maximum number of bytes per disk slice (SETUP-*.BIN file). Normally, this should be set to the total number of bytes available on the disk media divided by the value of the SlicesPerDisk [Setup] section directive, which defaults to 1.

This directive is ignored if disk spanning is not enabled using the DiskSpanning [Setup] section directive.

To optimally fill 4.7 GB recordable DVDs, use:

```
SlicesPerDisk=3
DiskSliceSize=1566000000
```

To optimally fill 8.5 GB (dual-layer) recordable DVDs, use:

SlicesPerDisk=5 DiskSliceSize=1708200000

To optimally fill 700 MB (80-minute) recordable CDs, use:

```
SlicesPerDisk=1
DiskSliceSize=736000000
```

To optimally fill 1.44MB floppy disks, use:

SlicesPerDisk=1 DiskSliceSize=1457664

[Setup]: DiskSpanning

Valid values: <u>yes or no</u>

Default value: no

Description:

If set to yes, the disk spanning feature will be enabled. Instead of storing all the compressed file data inside SETUP.EXE, the compiler will split it into multiple SETUP-*.BIN files -- known as "slices" -- suitable for copying onto separate floppy disks, CD-ROMs, or DVD-ROMs. Each generated slice contains a number in its name which indicates the disk onto which it should be copied. (For example, SETUP-2.BIN should be placed on disk 2.) The generated SETUP.EXE always goes on disk 1 along with the SETUP-1*.BIN file.

The size of each slice and the number of slices to create for each disk are determined by the values of the <u>DiskSliceSize</u> and <u>SlicesPerDisk</u> [Setup] section directives, respectively. Other disk spanning-related directives that you may want to tweak include <u>DiskClusterSize</u> and <u>ReserveBytes</u>.

Note that it is required that you set this directive to yes if the compressed size of your installation exceeds 2,100,000,000 bytes, even if you don't intend to place the installation onto multiple disks. (The installation will still function correctly if all the SETUP-*.BIN files are placed on the same disk.)

[Setup]: DontMergeDuplicateFiles

Valid values: yes or no

Description:

Obsolete in 4.2.5. Use MergeDuplicateFiles instead.

MergeDuplicateFiles=no is equivalent to DontMergeDuplicateFiles=yes.

[Setup]: EnableDirDoesntExistWarning

Valid values: yes or no

Default value: no

Description:

When set to yes, Setup will display a message box if the directory the user selects doesn't exist. Usually you will also set DirExistsWarning=no when this is yes.

[Setup]: Encryption

Valid values: <u>yes or no</u>

Default value: no

Description:

If set to yes, files that are compiled into the installation (via [Files] section entries) will be encrypted using ARCFOUR encryption, with a 160-bit key derived from the value of the <u>Password</u> [Setup] section directive.

Because of encryption import/export laws in some countries, encryption support is not included in the main Inno Setup installer and must be downloaded and installed separately if you wish to use it. See the <u>Inno Setup Downloads</u> page for more information.

If encryption is enabled and you call the <u>ExtractTemporaryFile</u> function from the [Code] section prior to the user entering the correct password, the function will fail unless the noencryption flag is used on the [Files] section entry for the file.

The key used for encryption is a 160-bit SHA-1 hash of 64-bit random salt plus the value of <u>Password</u>.

[Setup]: ExtraDiskSpaceRequired

Default value: 0

Description:

Normally, the disk space requirement displayed on the wizard is calculated by adding up the size of all the files in the [Files] section. If you want to increase the disk space display for whatever reason, set ExtraDiskSpaceRequired to the amount of bytes you wish to add to this figure. (1048576 bytes = 1 megabyte)

[Setup]: FlatComponentsList

Valid values: yes or no

Default value: yes

Description:

When this directive is set to yes, Setup will use 'flat' checkboxes for the components list. Otherwise Setup will use '3D' checkboxes.

[Setup]: InfoAfterFile

Description:

Specifies the name of an optional "readme" file, in .txt or .rtf (rich text) format, which is displayed after a successful install. This file must be located in your installation's <u>source directory</u> when running the Setup Compiler, unless a fully qualified pathname is specified or the pathname is prefixed by "compiler:", in which case it looks for the file in the Compiler directory.

This differs from isreadme files in that this text is displayed as a page of the wizard, instead of in a separate Notepad window.

If the user selects a language for which the InfoAfterFile parameter is set, this directive is effectively ignored. See the [Languages] section documentation for more information.

Example:

InfoAfterFile=infoafter.txt

[Setup]: InfoBeforeFile

Description:

Specifies the name of an optional "readme" file, in .txt or .rtf (rich text) format, which is displayed before the user selects the destination directory for the program. This file must be located in your installation's <u>source directory</u> when running the Setup Compiler, unless a fully qualified pathname is specified or the pathname is prefixed by "compiler:", in which case it looks for the file in the Compiler directory.

If the user selects a language for which the InfoBeforeFile parameter is set, this directive is effectively ignored. See the [Languages] section documentation for more information.

Example:

InfoBeforeFile=infobefore.txt

[Setup]: InternalCompressLevel

Valid values: none, or one of the LZMA compression levels

Default value: normal

Description:

This specifies the level of LZMA compression to use on Setup's internal structures. Generally, there is little reason to change this from the default setting of normal.

[Setup]: LanguageDetectionMethod

Valid values: uilanguage, locale, none

Default value: uilanguage

Description:

When set to uilanguage, Setup will determine the default language to use by checking the user's "UI language" (by calling GetUserDefaultUILanguage(), or on Windows versions where that function is unsupported, by reading the registry). This is the method that Microsoft recommends. The "UI language" is the language used in Windows' own dialogs. Thus, on an English edition of Windows, English will be the default, while on a Dutch edition of Windows, Dutch will be the default. On the MUI edition of Windows, the default will be the currently selected UI language.

When set to locale, Setup will determine the default language to use by calling GetUserDefaultLangID(). This function returns the setting of "Your locale" in Control Panel's Regional Options. It should however be noted that the "Your locale" option is not intended to affect languages; it is only documented to affect "numbers, currencies, times, and dates".

When set to none, Setup will use the first language specified in the [Languages] section as the default language.

[Setup]: LicenseFile

Description:

Specifies the name of an optional license agreement file, in .txt or .rtf (rich text) format, which is displayed before the user selects the destination directory for the program. This file must be located in your installation's <u>source directory</u> when running the Setup Compiler, unless a fully qualified pathname is specified or the pathname is prefixed by "compiler:", in which case it looks for the file in the Compiler directory.

If the user selects a language for which the LicenseFile parameter is set, this directive is effectively ignored. See the [Languages] section documentation for more information.

Example:

LicenseFile=license.txt

[Setup]: LZMAAlgorithm

Valid values: 0 or 1

Default value: 0 if the <u>LZMA compression level</u> is set to fast 1 otherwise

Description:

Controls the algorithm used by the LZMA and LZMA2 compressors.

A value of 0 enables the fast algorithm.

A value of 1 enables the normal algorithm.

[Setup]: LZMABlockSize

Valid values: 1024 through 262144

Default value: 4 * LZMADictionarySize

Description:

Controls the block size used by the LZMA2 compressor, in kilobytes, when <u>LZMANumBlockThreads</u> is set to 2 or higher.

Note that setting this too high can negate the benefits of using multiple block threads. Typically, the block size should be no more than the total size of your data divided by the number of block threads.

See also: <u>LZMADictionarySize</u> <u>LZMANumBlockThreads</u>

[Setup]: LZMADictionarySize

Valid values:	4 through 131072 (by default) 4 through 262144 if <u>LZMAUseSeparateProcess</u> is set to yes and running on 64-bit Windows (x64)
Default value:	32 if the LZMA compression level is set to fast 2048 if the LZMA compression level is set to normal 8192 if the LZMA compression level is set to max 32768 if the LZMA compression level is set to ultra 65536 if the LZMA compression level is set to ultra64

Description:

Controls the dictionary size used by the LZMA and LZMA2 compressors, in kilobytes. A larger dictionary size can provide a better compression ratio at the expense of compression speed and memory requirements.

Review the memory requirements listed in the <u>Compression</u> topic before using.

If an "Out of memory" error is seen when a very large dictionary size is used, <u>LZMAUseSeparateProcess</u> may need to be set.

See also:

LZMABlockSize

[Setup]: LZMAMatchFinder

Valid values: HC or BT

Default value: HC if the <u>LZMA compression level</u> is set to fast BT otherwise

Description:

Controls the match finder method used by the LZMA and LZMA2 compressors.

A value of HC enables the Hash Chain method with 4 hash bytes.

A value of BT enables the Binary Tree method with 4 hash bytes.

The Binary Tree method can provide a better compression ratio at the expense of compression speed.
[Setup]: LZMANumBlockThreads

Valid values: 1 through 32

Default value: 1

Description:

When compressing a large amount of data, the LZMA2 compressor has the ability to divide the data into "blocks" and compress two or more of these blocks in parallel through the use of additional threads (provided sufficient processor power is available). This directive specifies the number of threads to use -- that is, the maximum number of blocks that the LZMA2 compressor may compress in parallel.

The memory required during compression when multiple block threads are used is roughly:

LZMANumBlockThreads * (<u>Normal memory usage</u> + (<u>LZMABlockSize</u> * 2))

Since LZMA2 (and LZMA) uses two threads for match-finding by default (see <u>CompressionThreads</u>), there ideally should be two processor cores available for each block thread. Thus, to see the maximum benefit from a value of 2, four cores are needed.

Dividing the data into multiple blocks can reduce the compression ratio, as the compressor cannot find matches across blocks. Using a large <u>block size</u> can help to mitigate this.

If an "Out of memory" error is seen when multiple block threads are enabled in combination with a compression level that uses a large dictionary size (such as ultra64), <u>LZMAUseSeparateProcess</u> may need to be set.

See also: LZMABlockSize

[Setup]: LZMANumFastBytes

Valid values:	5 through 273
Default value:	64 if the <u>LZMA compression level</u> is set to max, ultra, or ultra64 32 otherwise

Description:

Controls number of fast bytes used by the LZMA and LZMA2 compressors. A larger number of fast bytes can provide a better compression ratio at the expense of compression speed.

[Setup]: LZMAUseSeparateProcess

Valid values: <u>yes, no</u>, or x86

Default value: no

Description:

Controls whether LZMA compression is performed inside the main compiler process or in a separate process.

Using a separate process for LZMA compression allows the compressor to allocate larger amounts of memory, which makes it possible for higher <u>LZMADictionarySize</u> and <u>LZMANumBlockThreads</u> settings to be used. Additionally, on 64-bit Windows (x64), a small increase in compression speed may be observed.

On 64-bit Windows (x64), there are no limitations on the amount of memory the compressor may use, as it runs inside a native 64-bit process. On 32-bit Windows, however, due to address space constraints, typically only about 1.5 GB is available for use by the compressor.

A value of yes enables the use of a 64-bit process on 64-bit Windows (x64), and a 32-bit process on 32-bit Windows.

A value of x86 enables the use of a 32-bit process only (normally only useful for debugging purposes).

A value of no disables the use of a separate process for LZMA compression.

Note that this directive only affects the compression of files specified in the [Files] section; compression of Setup's internal structures is always performed inside the main compiler process.

[Setup]: MergeDuplicateFiles

Valid values: <u>yes or no</u>

Default value: yes

Description:

Normally two file entries referring to the same source file will be compressed and stored only once. If you have a bunch of identical files in your installation, make them point to the same source file in the script, and the size of your installation can drop significantly. If you wish to disable this feature for some reason, set this directive to no.

[Setup]: MessagesFile

Description:

Obsolete in 4.0. This directive is no longer supported. Use the new [Languages] section to specify a custom messages file.

[Setup]: MinVersion

Format: *major.minor*

Default value: 5.0

Description:

This directive lets you specify a minimum <u>version of Windows</u> that your software runs on. <u>Build numbers and/or service pack levels</u> may be included.

If the user's system does not meet the minimum version requirement, Setup will give an error message and exit.

For compatibility with previous versions of Inno Setup, separate Windows 95/98/Me and Windows NT version numbers may be specified, separated by a comma. Example: MinVersion=0, 5.0. The Windows 95/98/Me version number (the first number) must be 0, however, as Inno Setup no longer supports Windows 95/98/Me.

[Setup]: OnlyBelowVersion

Format: *major.minor*

Default value: 0

Description:

This directive lets you specify a minimum <u>version of Windows</u> that your software *will not* run on. Specifying "0" means there is no upper version limit. <u>Build numbers and/or service pack levels</u> may be included.

This directive is essentially the opposite of MinVersion.

For compatibility with previous versions of Inno Setup, separate Windows 95/98/Me and Windows NT version numbers may be specified, separated by a comma. Example:

OnlyBelowVersion=0, 6.0. The Windows 95/98/Me version number (the first number) isn't used, however, as Inno Setup no longer supports Windows 95/98/Me.

[Setup]: Output

Valid values: yes or no

Default value: yes

Description:

If set to no the Setup Compiler will only check the script for errors and skip creating setup files.

[Setup]: OutputBaseFilename

Default value: setup

Description:

This directive allows you to assign a different name for the resulting Setup file(s), so you don't have to manually rename them after running the Setup Compiler.

Example:

OutputBaseFilename=MyProg100

[Setup]: OutputDir

Default value: Output

Description:

Specifies the "output" directory for the script, which is where the Setup Compiler will place the resulting SETUP.* files. By default, it creates a directory named "Output" under the directory containing the script for this.

If OutputDir is not a fully-qualified pathname, it will be treated as being relative to SourceDir, unless the pathname is prefixed by "userdocs:", in which case it will be treated as being relative to the the My Documents folder of the currenlty logged-in user. Setting OutputDir to . will result in the files being placed in the source directory.

Example:

OutputDir=c:\output

[Setup]: OutputManifestFile

Description:

When this directive is set, the compiler will create a manifest file detailing information about the files compiled into Setup. The file will be created in the <u>output directory</u> unless a path is included.

Example:

OutputManifestFile=Setup-Manifest.txt

[Setup]: Password

Description:

Specifies a password you want to prompt the user for at the beginning of the installation.

When using a password, you might consider setting <u>Encryption</u> to yes as well, otherwise files will be stored as plain text and it would not be exceedingly difficult for someone to gain access to them through reverse engineering.

The password itself is not stored as clear text; it's stored as a 160-bit SHA-1 hash, salted with a 64-bit random number. (Note: When encryption is enabled, this stored hash is *not* used for the encryption key; a different hash with a different salt is generated for that.)

[Setup]: PrivilegesRequired

Valid values: poweruser, admin, or lowest

Default value: admin

Description:

The effect of this directive depends on which version of Windows the user is running:

On Windows Vista and later:

This directive affects whether elevated rights are requested (via a User Account Control dialog) when the installation is started.

When set to admin (the default) or poweruser, Setup will always run with administrative privileges. If Setup was started by an unprivileged user, Windows will ask for the password to an account that has administrative privileges, and Setup will then run under that account.

When set to lowest, Setup will not request to be run with administrative privileges even if it was started by a member of the Administrators group. Additionally, the uninstall info root key will always be HKEY_CURRENT_USER, and the "common" forms of the Shell Folder constants are mapped to the "user" forms, even if administrative privileges are available. Do not use this setting unless you are sure your installation will run successfully on unprivileged accounts.

On earlier versions of Windows:

This directive specifies the minimum user privileges required to run the installation.

When set to admin (the default), Setup will only run if the user is a member of the Administrators group. Otherwise, it will display the following message and exit: "You must be logged in as an administrator when installing this program."

When set to poweruser, Setup will only run if the user is a member of the Administrators or Power Users groups. Otherwise, it will display the following message and exit: "You must be logged in as an administrator or as a member of the Power Users group when installing this program."

When set to lowest Setup will not check the user's group membership. Additionally, the uninstall info root key will always be HKEY_CURRENT_USER, and the "common" forms of the Shell Folder constants are mapped to the "user" forms, even if administrative privileges are available. Do not use this setting unless you are sure your installation will run successfully on unprivileged accounts.

Note:

Regardless of the version of Windows, only if the installation is per-user (PrivilegesRequired=lowest) you may touch Windows' per-user areas from your script.

[Setup]: ReserveBytes

Default value: 0

Description:

This specifies the minimum number of free bytes to reserve on the first disk. This is useful if you have to copy other files onto the first disk that aren't part of the setup program, such as a Readme file.

The Setup Compiler rounds this number up to the nearest cluster.

This directive is ignored if disk spanning is not enabled using the DiskSpanning [Setup] section directive.

[Setup]: RestartApplications

Valid values: <u>yes or no</u>

Default value: yes

Description:

When set to yes and <u>CloseApplications</u> is also set to yes, Setup restarts the closed applications after the installation has completed.

Note: For Setup to be able to restart an application after the installation has completed, the application needs to be using the Windows RegisterApplicationRestart API function.

See also: <u>CloseApplications</u> <u>CloseApplicationsFilter</u>

[Setup]: RestartIfNeededByRun

Valid values: <u>yes or no</u>

Default value: yes

Description:

When set to yes, and a program executed in the [Run] section queues files to be replaced on the next reboot (by calling MoveFileEx or by modifying wininit.ini), Setup will detect this and prompt the user to restart the computer at the end of installation.

[Setup]: SetupIconFile

Description:

Specifies a custom program icon to use for Setup/Uninstall. The file must be located in your installation's <u>source directory</u> when running the Setup Compiler, unless a fully qualified pathname is specified or the pathname is prefixed by "compiler:", in which case it looks for the file in the Compiler directory.

Example:

SetupIconFile=MyProgSetup.ico

[Setup]: SetupLogging

Valid values: <u>yes or no</u>

Default value: no

Description:

If set to yes, Setup will always create a log file. Equivalent to passing <u>/LOG</u> on the command line.

[Setup]: SetupMutex

Description:

This directive is used to prevent Setup from running while Setup is already running. It specifies the names of one or more named mutexes (multiple mutexes are separated by commas), which Setup will check for at startup. If any exist, Setup will display the message: "Setup has detected that Setup is currently running. Please close all instances of it now, then click OK to continue, or Cancel to exit." If none exist, Setup will create the mutex(es) and continue normally. The value may include constants.

To specify a mutex name containing a comma, escape the comma with a backslash.

See the topic for CreateMutex in the MS SDK help for more information on mutexes.

Example:

```
SetupMutex=MySetupsMutexName,Global\MySetupsMutexName
```

See also: <u>AppMutex</u>

[Setup]: ShowComponentSizes

Valid values: yes or no

Default value: yes

Description:

When this directive is set to yes, Setup will show the size of a component in the components list. Depending on the largest component, Setup will display sizes in kilobytes or in megabytes.

[Setup]: ShowLanguageDialog

Valid values: <u>yes, no</u>, or auto

Default value: yes

Description:

When set to yes and there are multiple [Languages] section entries, a *Select Language* dialog will be displayed to give the user an opportunity to override the language Setup chose by default. See the [Languages] section documentation for more information.

When set to no, the dialog will never be displayed.

When set to auto, the dialog will only be displayed if Setup does not find a language identifier match.

See also: <u>ShowUndisplayableLanguages</u> <u>UsePreviousLanguage</u>

[Setup]: ShowTasksTreeLines

Valid values: yes or no

Default value: no

Description:

When this directive is set to yes, Setup will show 'tree lines' between parent and sub tasks.

[Setup]: ShowUndisplayableLanguages

Valid values: <u>yes or no</u>

Default value: no

Description:

By default, languages that cannot be displayed on a user's system due to a code page mismatch are not listed in the *Select Language* dialog. For example, Russian text can only be displayed properly if the active code page is 1251; if the user isn't running code page 1251 they will not see Russian as an option.

If this directive is set to yes, all languages will be listed in the *Select Language* dialog. To avoid user confusion, it is not recommended that you enable this in production installs.

This directive is ignored by <u>Unicode Inno Setup</u>.

See also: <u>ShowLanguageDialog</u>

[Setup]: SignedUninstaller

Valid values: <u>yes or no</u>

Default value: yes if <u>SignTool</u> is set, no otherwise

Description:

Specifies whether the uninstaller program (unins???.exe) should be deployed with a digital signature attached. When the uninstaller has a valid digital signature, Windows Vista users will not see an "unidentified program" warning when launching it from outside of Control Panel.

The first time you compile a script with this directive set to yes, a uniquely-named copy of the uninstaller EXE file will be created in the directory specified by the <u>SignedUninstallerDir</u> directive (which defaults to the <u>output directory</u>). Depending on the <u>SignTool</u> setting, you will either then be prompted to attach a digital signature to this file using an external code-signing tool (such as Microsoft's signtool.exe) or the file will be automatically signed on the fly. On subsequent compiles, the signature from the file will be embedded into the compiled installations' uninstallers.

Upgrading to a newer version of Inno Setup, or changing certain [Setup] section directives that affect the contents of the uninstaller EXE file (such as <u>SetupIconFile</u>), will cause a new file to be created under a different name.

If a file generated by this directive is deleted, it will be recreated automatically if necessary on the next compile.

When the uninstaller has a digital signature, Setup will write the messages from the active language into a separate file (unins???.msg). It cannot embed the messages into the EXE file because doing so would invalidate the digital signature.

When set to yes, any temporary self-copies used by Setup are digitally signed too.

Details on obtaining signing certificates and using code-signing tools are beyond the scope of this documentation.

[Setup]: SignedUninstallerDir

Default value: <u>OutputDir</u>

Description:

Specifies the directory in which <u>signed uninstaller</u> files should be stored. By default, such files are stored in the <u>output directory</u>.

Separate script files may share the same SignedUninstallerDir setting. By setting up a common directory to hold signed uninstaller files, you won't have to re-sign the uninstaller each time you compile a new script file with a distinct OutputDir setting.

If SignedUninstallerDir is not a fully-qualified pathname, it will be treated as being relative to SourceDir. Setting SignedUninstallerDir to . will result in the files being placed in the source directory.

Example:

SignedUninstallerDir=c:\signeduninstallers

[Setup]: SignTool

Valid values: A name followed by zero or more parameters, space separated

Description:

Specifies the name and parameters of the Sign Tool to be used to digitally sign Setup (and Uninstall if <u>SignedUninstaller</u> is set to yes). When Setup has a valid digital signature, users will not see an "unidentified program" warning when launching it.

The specified Sign Tool name and its command have to be defined in the compiler IDE (via the *Tools* | *Configure Sign Tools...* menu) or on the <u>compiler command line</u> (via the "/S" parameter), else an error will occur.

The following special sequences may be used in Sign Tool parameters and commands:

\$f, replaced by the quoted file name of the file to be signed.

\$p, replaced by the Sign Tool parameters.

\$q, replaced by a quote, useful for defining a Sign Tool which contains quotes from the command line.

\$\$, replaced by a single \$ character.

Example:

Assume the following Sign Tools have been defined in the IDE:

```
mystandard=signtool.exe /x /y /d $qMy Program$q $f
mycustom=signtool.exe $p
byparam=$p
```

then some examples would be:

```
SignTool=mystandard
SignTool=mycustom /x /y /d $qMy Program$q $f
SignTool=byparam signtool.exe /x /y /d $qMy Program$(
```

Note: for security reasons you should give a unique name to any Sign

Tool set to \$p, and not use a byparam name copied from this example. Consider what happens if you #include a third-party file that says:

Þ

```
SignTool=byparam format c:
```

Details on obtaining signing certificates and using code-signing tools are beyond the scope of this documentation.

Note: If you use a Sign Tool and your Setup contains a large amount of data, it is recommended that you enable <u>Disk spanning</u> with <u>DiskSliceSize</u> set to max. If you don't do this, the user might experience a long delay after starting Setup caused by Windows verifying the digital signature against all your data. There should be no security reduction from using disk spanning in practice: all files extracted from the unsigned .bin files undergo SHA-1 verification (provided dontverifychecksum isn't used). The SHA-1 hashes for this (along with all other metadata) are kept inside Setup's EXE, which is protected by the digital signature.

See also: <u>SignToolRetryCount</u>

[Setup]: SignToolRetryCount

Default value: 2

Description:

Specifies the number of times the Setup Compiler should automatically retry digital signing on any errors.

See also: SignTool

[Setup]: SlicesPerDisk

Valid values: 1 through 26

Default value: 1

Description:

The number of SETUP-*.BIN files to create for each disk. If this is 1 (the default setting), the files will be named SETUP-*x*.BIN, where *x* is the disk number. If this is greater than 1, the files will be named SETUP-*xy*.BIN, where *x* is the disk number and *y* is a unique letter.

One reason why you may need to increase this from the default value of 1 is if the size of your disk media exceeds 2,100,000,000 bytes -- the upper limit of the DiskSliceSize [Setup] section directive. If, for example, your disk media has a capacity of 3,000,000,000 bytes, you can avoid the 2,100,000,000-byte disk slice size limit by setting SlicesPerDisk to 2 and DiskSliceSize to 1500000000 (or perhaps slightly less, due to file system overhead).

[Setup]: SolidCompression

Valid values: <u>yes or no</u>

Default value: no

Description:

If yes, solid compression will be enabled. This causes all files to be compressed at once instead of separately. This can result a much greater overall compression ratio if your installation contains many files with common content, such as text files, especially if such common content files are grouped together within the [Files] section.

The disadvantage to using solid compression is that because all files are compressed into a single compressed stream, Setup can no longer randomly access the files. This can decrease performance. If a certain file isn't going to be extracted on the user's system, it has to decompress the data for that file anyway (into memory) before it can decompress the next file. And if, for example, there was an error while extracting a particular file and the user clicks Retry, it can't just seek to the beginning of that file's compressed data; since all files are stored in one stream, it has seek to the very beginning. If disk spanning was enabled, the user would have to re-insert disk 1.

Thus, it is not recommended that solid compression be enabled on huge installs (say, over 100 MB) or on disk-spanned installs. It is primarily designed to save download time on smaller installs distributed over the Internet.

[Setup]: SourceDir

Description:

Specifies a new <u>source directory</u> for the script.

Example:

SourceDir=c:\files

[Setup]: TerminalServicesAware

Valid values: yes or no

Default value: yes

Description:

Specifies whether the compiler should set the "Terminal Services aware" flag in the headers of the Setup and Uninstall programs. This feature is new to version 5.1.7 and defaults to yes; previous versions never set the flag.

Most importantly, the "Terminal Services aware" flag affects the behavior of the {win} constant (and <u>GetWinDir</u> support function) on servers with Terminal Services installed in application mode.

When the flag is set, {win} will consistently return the system's real Windows directory, typically "C:\WINDOWS", just as on systems that do not have Terminal Services installed.

When the flag is not set, Windows runs the program in compatibility mode, where {win} may return either the real Windows directory or a user-specific Windows directory, such as "C:\Documents and Settings\ <user name>\WINDOWS". Which one you get depends on the name of the program's EXE file and how it is launched. If the program is named setup.exe or install.exe, or if it is launched from the *Add/Remove Programs* Control Panel applet, then Windows will put the system in "install mode", which effectively makes the program (and all other programs running in the session) behave as if the "Terminal Services aware" flag were set. Otherwise, the program is treated as a legacy application and is given a private Windows directory. (This is true even if the user running the program has full administrative privileges.)

Because the behavior that results from setting

TerminalServicesAware to no is inconsistent and hard to predict, it is recommended that you use the default setting of yes. Only use no as a temporary fix if you encounter troubles on systems with Terminal Services after upgrading from a previous Inno Setup version.

[Setup]: TimeStampRounding

Valid values: 0 through 60

Default value: 2

Description:

By default, time stamps on files referenced by [Files] section entries are rounded down to the nearest 2-second boundary. FAT partitions have only a 2-second time stamp resolution, so this ensures that time stamps are set the same way on both FAT and NTFS partitions.

The rounding can be altered or disabled by setting the TimeStampRounding directive. Setting it to 0 will disable the rounding. Setting it to a number between 1 and 60 will cause time stamps to be rounded down to the nearest TimeStampRounding-second boundary.

[Setup]: TimeStampsInUTC

Valid values: <u>yes or no</u>

Default value: no

Description:

By default, time stamps on files referenced by [Files] section entries are saved and restored as local times. This means that if a particular file has a time stamp of 01:00 local time at compile time, Setup will extract the file with a time stamp of 01:00 local time, regardless of the user's time zone setting or whether DST is in effect.

If TimeStampsInUTC is set to yes, time stamps will be saved and restored in UTC -- the native time format of Win32 and NTFS. In this mode, a file with a time stamp of 01:00 local time in New York will have a time stamp of 06:00 local time when installed in London.

[Setup]: TouchDate

Valid values: current, none, or YYYY-MM-DD

Default value: current

Description:

The date used in the time/date stamp of files referenced by [Files] section entries that include the touch flag.

A value of current causes the current system date (at compile time) to be used. A value of none leaves the date as-is. Otherwise, TouchDate is interpreted as an explicit date in *YYYY-MM-DD* (ISO 8601) format. If <u>TimeStampsInUTC</u> is set to yes, the date is assumed to be a UTC date.

Example:

TouchDate=2004-01-31
[Setup]: TouchTime

Valid values: current, none, HH: MM, or HH: MM: SS

Default value: current

Description:

The time used in the time/date stamp of files referenced by [Files] section entries that include the touch flag.

A value of current causes the current system time (at compile time) to be used. A value of none leaves the time as-is. Otherwise, TouchTime is interpreted as an explicit time in *HH*:*MM* or *HH*:*MM*:*SS* format. If <u>TimeStampsInUTC</u> is set to yes, the time is assumed to be a UTC time.

Example:

TouchTime=13:00

[Setup]: Uninstallable

Valid values: yes or no, or a boolean expression

Default value: yes

Description:

This determines if Inno Setup's automatic uninstaller is to be included in the installation. If this is yes or to a <u>boolean expression</u> evaluating to True the uninstaller is included. Otherwise, no uninstallation support is included, requiring the end-user to manually remove the files pertaining to your application.

Setting this to a boolean expression can be useful if you want to offer the user a 'portable mode' option.

```
Example:
[Setup]
Uninstallable=not IsTaskSelected('portablemode')
[Tasks]
```

Name: portablemode; Description: "Portable Mode"

See also: CreateUninstallRegKey

[Setup]: UninstallDisplayIcon

Description:

This lets you specify a particular icon file (either an executable or an .ico file) to display for the Uninstall entry in the *Add/Remove Programs* Control Panel applet. The filename will normally begin with a directory constant.

If the file you specify contains multiple icons, you may append the suffix ",n" to specify an icon index, where n is the zero-based numeric index.

If this directive is not specified or is blank, Windows will select an icon itself, which may not be the one you prefer.

Examples:

UninstallDisplayIcon={app}\MyProg.exe
UninstallDisplayIcon={app}\MyProg.exe,1

[Setup]: UninstallDisplayName

Description:

This lets you specify a custom name for the program's entry in the *Add/Remove Programs* Control Panel applet. The value may include constants. If this directive is not specified or is blank, Setup will use the value of [Setup] section directive AppVerName for the name.

Due to limitations of Windows 95/98/Me's *Add/Remove Programs* Control Panel applet, the value of UninstallDisplayName will be trimmed if it exceeds 63 characters.

Example:

UninstallDisplayName=My Program

[Setup]: UninstallDisplaySize

Description:

On Windows 7 and newer, Setup uses this directive to set the EstimatedSize value in the Uninstall registry key when possible since the Windows 7 Add/Remove Programs Control Panel (called Program and Features) no longer automatically calculates it. If an UninstallDisplaySize is not set, Setup estimates the size itself by taking the size of all files installed and adding any ExtraDiskSpaceRequired values set. Note: Windows 7 without any service pack only supports the display of values smaller than 4 GB.

Set in bytes. (1048576 bytes = 1 megabyte)

Example:

UninstallDisplaySize=1073741824

[Setup]: UninstallFilesDir

Default value: {app}

Description:

Specifies the directory where the "unins*.*" files for the uninstaller are stored.

Note: You should not assign a different value here on a new version of an application, or else Setup won't find the uninstall logs from the previous versions and therefore won't be able to <u>append to</u> them.

Example:

UninstallFilesDir={app}\uninst

[Setup]: UninstallIconFile

Description:

Obsolete in 5.0.0. As Setup and Uninstall have been merged into a single executable, setting a custom icon for Uninstall is no longer possible.

[Setup]: UninstallIconName

Description:

Obsolete in 3.0. This directive is no longer supported. If you wish to create an Uninstall icon, use the new {uninstallexe} constant in the Filename parameter of an [lcons] section entry.

[Setup]: UninstallLogMode

Valid values: append, new, or overwrite

Default value: append

Description:

append, the default setting, instructs Setup to <u>append to an existing</u> <u>uninstall log</u> when possible.

new, which corresponds to the behavior in pre-1.3 versions of Inno Setup, instructs Setup to always create a new uninstall log.

overwrite instructs Setup to overwrite any existing uninstall logs from the <u>same application</u> instead of appending to them (this is *not* recommended). The same rules for appending to existing logs apply to overwriting existing logs.

Example:

UninstallLogMode=append

[Setup]: UninstallRestartComputer

Valid values: yes or no

Default value: no

Description:

When set to yes, the uninstaller will always prompt the user to restart the system at the end of a successful uninstallation, regardless of whether it is necessary (e.g., because of [Files] section entries with the uninsrestartdelete flag).

[Setup]: UninstallStyle

Description:

Obsolete in 5.0.0. Only the "modern" uninstaller style is supported now.

[Setup]: UpdateUninstallLogAppName

Valid values: yes or no

Default value: yes

Description:

If yes, when appending to an existing uninstall log, Setup will replace the AppName field in the log with the current installation's AppName. The AppName field of the uninstall log determines the title displayed in the uninstaller. You may want to set this to no if your installation is merely an upgrade or add-on to an existing program, and you don't want the title of the uninstaller changed.

[Setup]: UsePreviousAppDir

Valid values: <u>yes or no</u>

Default value: yes

Description:

When this directive is yes, the default, at startup Setup will look in the registry to see if the <u>same application</u> is already installed, and if so, it will use the directory of the previous installation as the default directory presented to the user in the wizard.

[Setup]: UsePreviousGroup

Valid values: <u>yes or no</u>

Default value: yes

Description:

When this directive is yes, the default, at startup Setup will look in the registry to see if the <u>same application</u> is already installed, and if so, it will use the Start Menu folder name of the previous installation as the default Start Menu folder name presented to the user in the wizard. Additionally, if AllowNoIcons is set to yes, the *Don't create a Start Menu folder* setting from the previous installation will be restored.

[Setup]: UsePreviousLanguage

Valid values: <u>yes or no</u>

Default value: yes

Description:

When this directive is yes, the default, at startup Setup will look in the registry to see if the <u>same application</u> is already installed, and if so, it will use the language of the previous installation as the default language selected in the list of available languages on the *Select Language* dialog.

Note that this directive does not change the language used by the *Select Language* dialog itself because it cannot assume that the current user understands the same languages as the previous user. See the [Languages] section help topic for details on which language the *Select Language* dialog uses by default.

Also note that Setup cannot re-use settings from a previous installation that had Uninstallable set to no, since the registry entries it looks for are not created when Uninstallable is no.

UsePreviousLanguage must be set to no when AppId includes constants.

[Setup]: UsePreviousSetupType

Valid values: yes or no

Default value: yes

Description:

When this directive is yes, the default, at startup Setup will look in the registry to see if the <u>same application</u> is already installed, and if so, it will use the setup type and component settings of the previous installation as the default settings presented to the user in the wizard.

[Setup]: UsePreviousTasks

Valid values: <u>yes or no</u>

Default value: yes

Description:

When this directive is yes, the default, at startup Setup will look in the registry to see if the <u>same application</u> is already installed, and if so, it will use the task settings of the previous installation as the default settings presented to the user in the wizard.

[Setup]: UsePreviousUserInfo

Valid values: yes or no

Default value: yes

Description:

When this directive is yes, the default, at startup Setup will look in the registry to see if the <u>same application</u> is already installed, and if so, it will use the name, organization and serial number entered previously as the default settings presented to the user on the *User Information* wizard page.

[Setup]: UserInfoPage

Valid values: <u>yes or no</u>

Default value: no

Description:

If this is set to yes, Setup will show a *User Information* wizard page which asks for the user's name, organization and possibly a serial number. The values the user enters are stored in the {userinfoname}, {userinfoorg} and {userinfoserial} constants. You can use these constants in [Registry] or [INI] entries to save their values for later use.

For the serial number field to appear, a <u>CheckSerial</u> event function must be present.

The <u>DefaultUserInfoName</u>, <u>DefaultUserInfoOrg</u> and <u>DefaultUserInfoSerial</u> directives determine the default name, organization and serial number shown. If <u>UsePreviousUserInfo</u> is yes (the default) and Setup finds that a previous version of the <u>same</u> <u>application</u> is already installed, it will use the name, organization and serial number entered previously instead.

On silent installs, the default user name, organization, and serial number values will be assumed. Setup will not check whether the user name is blank (since the user has no way of correcting it), however it will still check the serial number.

[Setup]: UseSetupLdr

Valid values: <u>yes or no</u>

Default value: yes

Description:

This tells the Setup Compiler which type of Setup to create. If this is yes, it compiles all setup data into a single EXE. If this is no, it compiles the setup data into at least three files: setup.exe, setup-0.bin, and setup-1.bin. The **only** reason you would probably want to use no is for debugging purposes.

Note: Do not use UseSetupLdr=no on an installation which uses disk spanning (DiskSpanning=yes). When UseSetupLdr is yes, the setup program is copied to and run from the user's TEMP directory. This does not happen when UseSetupLdr is no, and could result in errors if Windows tries to locate the setup.exe file on the disk and can't find it because a different disk is in the drive.

Note: Do not use UseSetupLdr=no to avoid digital signature verification startup delays on a large Setup, use disk spanning instead. See <u>SignTool</u> for more information. Also note that digitally signing a UseSetupLdr=no based Setup will lead to an invalid digital signature for Uninstall.

[Setup]: VersionInfoCompany

Default value: <u>AppPublisher</u> if <u>AppPublisher</u> doesn't include constants, an empty string otherwise

Description:

Specifies the company name value for the Setup version info.

[Setup]: VersionInfoCopyright

Default value: AppCopyright if AppCopyright doesn't include constants, an empty string otherwise

Description:

Specifies the copyright value for the Setup version info.

[Setup]: VersionInfoDescription

Default value: "<u>AppName</u> Setup" if <u>AppName</u> doesn't include constants, an empty string otherwise

Description:

Specifies the file description value for the Setup version info.

[Setup]: VersionInfoProductName

Default value: AppName if AppName doesn't include constants, an empty string otherwise

Description:

Specifies the product name value for the Setup version info.

[Setup]: VersionInfoProductTextVersion

Default value: VersionInfoProductVersion if set, else AppVersion if set and does not include constants, else VersionInfoTextVersion

Description:

Specifies the textual product version value for the Setup version info.

[Setup]: VersionInfoProductVersion

Valid values: A value in the form of up to 4 numbers separated by dots

Default value: VersionInfoVersion

Description:

Specifies the binary product version value for the Setup version info.

Partial version numbers are allowed. Missing numbers will be appended as zero's.

Note that this value is only known to be displayed by Explorer on Windows Vista SP2. Other versions display the textual product version value (<u>VersionInfoProductTextVersion</u>) instead.

[Setup]: VersionInfoTextVersion

Default value: VersionInfoVersion

Description:

Specifies the textual file version value for the Setup version info.

Note that this value was only displayed on Explorer's Version tab on Windows 98 and earlier. Later versions display the binary version value (<u>VersionInfoVersion</u>) instead.

[Setup]: VersionInfoVersion

Valid values: A value in the form of up to 4 numbers separated by dots

Default value: 0.0.0.0

Description:

Specifies the binary file version value for the Setup version info.

Partial version numbers are allowed. Missing numbers will be appended as zero's.

[Setup]: WindowResizable

Valid values: <u>yes or no</u>

Default value: yes

Description:

If set to no, the user won't be able to resize the Setup program's background window when it's not maximized.

This directive has no effect if WindowVisible is not set to yes.

[Setup]: WindowShowCaption

Valid values: <u>yes or no</u>

Default value: yes

Description:

If set to no, Setup will be truly "full screen" -- it won't have a caption bar or border, and it will be on top of the taskbar.

This directive has no effect if WindowVisible is not set to yes.

[Setup]: WindowStartMaximized

Valid values: yes or no

Default value: yes

Description:

If set to yes, the Setup program's background window will initially be displayed in a maximized state, where it won't cover over the taskbar.

This directive has no effect if WindowVisible is not set to yes.

[Setup]: WindowVisible

Valid values: <u>yes or no</u>

Default value: no

Description:

If set to yes, there will be a gradient background window displayed behind the wizard.

Note that this is considered a legacy feature; it likely will be removed at some point in the future.

[Setup]: WizardImageBackColor

Valid values: A value in the form of \$bbggrr, where rr, gg, and bb specify the two-digit intensities (in hexadecimal) for red, green, and blue respectively. Or it may be one of the following predefined color names: clBlack, clMaroon, clGreen, clOlive, clNavy, clPurple, clTeal, clGray, clSilver, clRed, clLime, clYellow, clBlue, clFuchsia, clAqua, clWhite.

Default value: \$400000

Description:

This directive specifies the background color used to fill any unused space around the wizard bitmap (which is specified by <u>WizardImageFile</u>). There can only be unused space if <u>WizardImageStretch</u> is set to no.

[Setup]: WizardImageFile

Default value: compiler:WIZMODERNIMAGE.BMP

Description:

Specifies the name of the bitmap file to display on the left side of the wizard in the Setup program. This file must be located in your installation's <u>source directory</u> when running the Setup Compiler, unless a fully qualified pathname is specified or the pathname is prefixed by "compiler:", in which case it looks for the file in the Compiler directory.

256-color bitmaps may not display correctly in 256-color mode, since it does not handle palettes. The maximum size of the bitmap is 164x314 pixels. Note that if Windows is running with Large Fonts, the area on the wizard for the bitmap will be larger.

Example:

WizardImageFile=myimage.bmp

[Setup]: WizardImageStretch

Valid values: <u>yes or no</u>

Default value: yes

Description:

If set to yes, the default, the wizard images will be stretched or shrunk if the wizard is larger or smaller than normal, e.g. if the user is running in Large Fonts.

If set to no, the wizard images will be centered in their respective areas if the wizard is larger than normal, and clipped if the wizard is smaller than normal. (This corresponds to the default behavior of Inno Setup 4.1.2 and earlier.)

[Setup]: WizardSmallImageBackColor

Description:

Obsolete in 5.0.4. This directive formerly specified the background color used to fill any unused space around the small wizard bitmap when <u>WizardImageStretch</u> was set to no. Now any unused space is filled with the standard window color (usually white). If you wish to create a colored border around the image, do so by modifying the bitmap itself.
[Setup]: WizardSmallImageFile

Default value: compiler:WIZMODERNSMALLIMAGE.BMP

Description:

Specifies the name of the bitmap file to display in the upper right corner of the wizard window. This file must be located in your installation's <u>source directory</u> when running the Setup Compiler, unless a fully qualified pathname is specified or the pathname is prefixed by "compiler:", in which case it looks for the file in the Compiler directory.

256-color bitmaps may not display correctly in 256-color mode, since it does not handle palettes. The maximum size of the bitmap is 55x58 pixels.

Example:

WizardSmallImageFile=mysmallimage.bmp

[Setup]: WizardStyle

Valid values: modern

Default value: modern

Description:

Obsolete in 3.0. Inno Setup 2.x supported an alternate wizard style called "classic". Support for the "classic" style has been dropped in Inno Setup 3.0.

[Types] section

This section is optional. It defines all of the setup types Setup will show on the *Select Components* page of the wizard. During compilation a set of default setup types is created if you define components in a [Components] section but don't define types. If you are using the default (English) messages file, these types are the same as the types in the example below.

Here is an example of a [Types] section:

[Types] Name: "full"; Description: "Full installation" Name: "compact"; Description: "Compact installati Name: "custom"; Description: "Custom installation

The following is a list of the supported parameters:

Name (Required)

The internal name of the type. Used as parameter for components in the [Components] section to instruct Setup to which types a component belongs.

Example:

Name: "full"

Description (Required)

The description of the type, which can include constants. This description is shown during installation.

```
Example:
```

Description: "Full installation"

Flags

This parameter is a set of extra options. Multiple options may be used by separating them by spaces. The following options are supported:

iscustom

Instructs Setup that the type is a custom type. Whenever the end user manually changes the components selection during installation, Setup will set the setup type to the custom type. Note that if you don't define a custom type, Setup will only allow the user to choose a setup type and he/she can no longer manually select/unselect components.

Only one type may include this flag.

Example:

Flags: iscustom

Common Parameters

[Components] section

This section is optional. It defines all of the components Setup will show on the *Select Components* page of the wizard for setup type customization.

By itself a component does nothing: it needs to be 'linked' to other installation entries. See <u>Components and Tasks Parameters</u>.

Here is an example of a [Components] section:

```
[Components]
Name: "main"; Description: "Main Files"; Types: f
Name: "help"; Description: "Help Files"; Types: f
Name: "help\english"; Description: "English"; Typ
Name: "help\dutch"; Description: "Dutch"; Types:
```

The example above generates four components: A "main" component which gets installed if the end user selects a type with name "full" or "compact" and a "help" component which has two child components and only gets installed if the end user selects the "full" type.

The following is a list of the supported parameters:

Name (Required)

The internal name of the component.

The total number of \ or / characters in the name of the component is called the level of the component. Any component with a level of 1 or more is a child component. The component listed before the child component with a level of 1 less than the child component, is the parent component. Other components with the same parent component as the child component are sibling components.

A child component can't be selected if its parent component isn't selected. A parent component can't be selected if none of its children are selected, unless a Component's parameter directly references the parent component or the parent component includes the checkablealone flag. If sibling components have the exclusive flag, only one of them can be selected.

Example: Name: "help"

Description (Required)

The description of the component, which can include constants. This description is shown to the end user during installation.

```
Example:
Description: "Help Files"
```

Types

A space separated list of types this component belongs to. If the end user selects a type from this list, this component will be installed.

If the fixed flag isn't used (see below), any custom types (types using the iscustom flag) in this list are ignored by Setup.

Example: Types: full compact

ExtraDiskSpaceRequired

The extra disk space required by this component, similar to the <u>ExtraDiskSpaceRequired</u> directive for the [Setup] section.

Example:

```
ExtraDiskSpaceRequired: 0
```

Flags

This parameter is a set of extra options. Multiple options may be used by separating them by spaces. The following options are supported:

checkablealone

Specifies that the component can be checked when none of its children are. By default, if no Component's parameter directly references the component, unchecking all of the component's children will cause the component to become unchecked.

dontinheritcheck

Specifies that the component should not automatically become checked when its parent is checked. Has no effect on top-level components, and cannot be combined with the exclusive flag.

exclusive

Instructs Setup that this component is mutually exclusive with sibling components that also have the exclusive flag.

fixed

Instructs Setup that this component can not be manually selected or unselected by the end user during installation.

restart

Instructs Setup to ask the user to restart the system if this component is installed, regardless of whether this is necessary (for example because of [Files] section entries with the restartreplace flag). Like <u>AlwaysRestart</u> but per component.

disablenouninstallwarning

Instructs Setup not to warn the user that this component will not be uninstalled after he/she deselected this component when it's already installed on his/her machine.

Depending on the complexity of your components, you can try to use the [InstallDelete] section and this flag to automatically 'uninstall' deselected components.

Example:

Flags: fixed

Common Parameters

[Tasks] section

This section is optional. It defines all of the user-customizable tasks Setup will perform during installation. These tasks appear as check boxes and radio buttons on the *Select Additional Tasks* wizard page.

By itself a task does nothing: it needs to be 'linked' to other installation entries. See <u>Components and Tasks Parameters</u>.

Here is an example of a [Tasks] section:

```
[Tasks]
Name: desktopicon; Description: "Create a &deskto
Name: desktopicon\common; Description: "For all u
Name: desktopicon\user; Description: "For the cur
Name: quicklaunchicon; Description: "Create a &Qu
Name: associate; Description: "&Associate files";
```

The following is a list of the supported parameters:

Name (Required)

The internal name of the task.

The total number of \ or / characters in the name of the task is called the level of the task. Any task with a level of 1 or more is a child task. The task listed before the child task with a level of 1 less than the child task, is the parent task. Other tasks with the same parent task as the child task are sibling tasks.

A child task can't be selected if its parent task isn't selected. A parent task can't be selected if none of its children are selected, unless a Tasks parameter directly references the parent task or the parent task includes the checkablealone flag.

If sibling tasks have the exclusive flag, only one of them can be selected.

```
Example:
Name: "desktopicon"
```

Description (Required)

The description of the task, which can include constants. This description is shown to the end user during installation.

```
Example:
Description: "Create a &desktop icon"
```

GroupDescription

The group description of a group of tasks, which can include constants. Consecutive tasks with the same group description will be grouped below a text label. The text label shows the group description.

Example:

GroupDescription: "Additional icons"

Components

A space separated list of components this task belongs to. If the end user selects a component from this list, this task will be shown. A task entry without a Components parameter is always shown.

Example:

Components: main

Flags

This parameter is a set of extra options. Multiple options may be used by separating them by spaces. The following options are supported:

checkablealone

Specifies that the task can be checked when none of its children are. By default, if no Tasks parameter directly references the task, unchecking all of the task's children will cause the task to become unchecked.

checkedonce

Instructs Setup that this task should be unchecked initially when Setup finds a previous version of the <u>same application</u> is already installed.

If the UsePreviousTasks [Setup] section directive is no, this flag is effectively disabled.

dontinheritcheck

Specifies that the task should not automatically become checked when its parent is checked. Has no effect on top-level tasks, and cannot be combined with the exclusive flag.

exclusive

Instructs Setup that this task is mutually exclusive with sibling tasks that also have the exclusive flag.

restart

Instructs Setup to ask the user to restart the system at the end of installation if this task is selected, regardless of whether it is necessary (for example because of [Files] section entries with the restartreplace flag). Like <u>AlwaysRestart</u> but per task.

unchecked

Instructs Setup that this task should be unchecked initially.

Example:

Flags: unchecked

Common Parameters

[Dirs] section

This optional section defines any additional directories Setup is to create *besides* the application directory the user chooses, which is created automatically. Creating subdirectories underneath the main application directory is a common use for this section.

Note that you aren't required to explicitly create directories before installing files to them using the [Files] section, so this section is primarily useful for creating empty directories.

Here is an example of a [Dirs] section:

```
[Dirs]
Name: "{app}\data"
Name: "{app}\bin"
```

The example above will, after Setup creates the application directory, create two subdirectories underneath the application directory.

The following is a list of the supported parameters:

Name (Required)

The name of the directory to create, which normally will start with one of the directory constants.

```
Example:
Name: "{app}\MyDir"
```

Attribs

Specifies additional attributes for the directory. This can include one or more of the following: readonly, hidden, system. If this parameter is not specified, Setup does not assign any special attributes to the directory.

If the directory already exists, the specified attributes will be combined with the directory's existing attributes.

Example:

Permissions

Specifies additional permissions to grant in the directory's ACL (access control list). It is not recommended that you use this parameter if you aren't familiar with ACLs or why you would need to change them, because misusing it could negatively impact system security.

For this parameter to have an effect the directory must be located on a partition that supports ACLs (such as NTFS), and the current user must be able to change the permissions on the directory. In the event these conditions are not met, no error message will be displayed, and the permissions will not be set.

This parameter should *only* be used on directories private to your application. Never change the ACLs on top-level directories like {sys} or {pf}, otherwise you can open up security holes on your users' systems.

In addition, it is recommended that you avoid using this parameter to grant write access on directories containing program files. Granting, for example, everyone-modify permission on the {app} directory will allow unprivileged users to tamper with your application's program files; this creates the potential for a privilege escalation vulnerability. (However, it is safe to change the permissions on a subdirectory of your application's directory which does not contain program files, e.g. {app}\data.)

The specified permissions are set regardless of whether the directory existed prior to installation.

This parameter can include one or more space separated values in the format:

```
<user or group identifier>-<access type>
```

The following access types are supported for the [Dirs] section:

full

Grants "Full Control" permission, which is the same as modify (see below), but additionally allows the specified user/group to take ownership of the directory and change its permissions. Use sparingly; generally, modify is sufficient.

modify

Grants "Modify" permission, which allows the specified user/group to read, execute, create, modify, and delete files in the directory and its subdirectories.

readexec

Grants "Read & Execute" permission, which allows the specified user/group to read and execute files in the directory and its subdirectories.

Example:

Permissions: users-modify

Flags

This parameter is a set of extra options. Multiple options may be used by separating them by spaces. The following options are supported:

deleteafterinstall

Instructs Setup to create the directory as usual, but then delete it once the installation is completed (or aborted) if it's empty. This can be useful when extracting temporary data needed by a program executed in the script's [Run] section.

This flag will not cause directories that already existed before installation to be deleted.

setntfscompression

Instructs Setup to enable NTFS compression on the directory. If it fails to set the compression state for any reason (for example, if compression is not supported by the file system), no error message will be displayed.

If the directory already exists, the compression state of any

files present in the directory will not be changed.

uninsalwaysuninstall

Instructs the uninstaller to always attempt to delete the directory if it's empty. Normally the uninstaller will only try to delete the directory if it didn't already exist prior to installation.

uninsneveruninstall

Instructs the uninstaller to not attempt to delete the directory. By default, the uninstaller deletes any directory specified in the [Dirs] section if it is empty.

unsetntfscompression

Instructs Setup to disable NTFS compression on the directory. If it fails to set the compression state for any reason (for example, if compression is not supported by the file system), no error message will be displayed.

If the directory already exists, the compression state of any files present in the directory will not be changed.

Example:

Flags: uninsneveruninstall

Components and Tasks Parameters

Common Parameters

[Files] section

This optional section defines any files Setup is to install on the user's system.

Here is an example of a [Files] section:

```
[Files]
Source: "CTL3DV2.DLL"; DestDir: "{sys}"; Flags: o
Source: "MYPROG.EXE"; DestDir: "{app}"
Source: "MYPROG.CHM"; DestDir: "{app}"
Source: "README.TXT"; DestDir: "{app}"; Flags: is
```

See the *Remarks* section at the bottom of this topic for some important notes.

The following is a list of the supported parameters:

Source (Required)

The name of the *source file*. The compiler will prepend the path of your installation's <u>source directory</u> if you do not specify a fully qualified pathname.

This can be a wildcard to specify a group of files in a single entry. When a wildcard is used, all files matching it use the same options.

When the flag external is specified, Source must be the full pathname of an existing file (or wildcard) on the distribution media or the user's system (e.g. "{src}\license.ini").

Constants may only be used when the external flag is specified, because the compiler does not do any constant translating itself.

```
Examples:
Source: "MYPROG.EXE"
Source: "Files\*"
```

DestDir (Required)

The directory where the file is to be installed on the user's system.

Will almost always begin with one of the directory constants. If the specified path does not already exist on the user's system, it will be created automatically, and removed automatically during uninstallation if empty.

```
Examples:
DestDir: "{app}"
DestDir: "{app}\subdir"
```

DestName

This parameter specifies a new name for the file when it is installed on the user's system. By default, Setup uses the name from the Source parameter, so in most cases it's not necessary to specify this parameter.

```
Example:
DestName: "MYPROG2.EXE"
```

Excludes

Specifies a list of patterns to exclude, separated by commas. This parameter cannot be combined with the external flag.

Patterns may include wildcard characters ("*" and "?"). Note that unlike the Source parameter, a simple Unix-style pattern matching routine is used for Excludes. Dots in the pattern are always significant, thus "*.*" will not exclude a file with no extension (instead, use just "*"). Also, question marks always match exactly one character, thus "?????" will not exclude files with names less than five characters long.

If a pattern starts with a backslash ("\") it is matched against the start of a path name, otherwise it is matched against the end of a path name. Thus "\foo" will only exclude a file named "foo" at the base of the tree. On the other hand, "foo" will exclude any file named "foo" anywhere in the tree.

The patterns may include backslashes. "foo\bar" will exclude both

"foo\bar" and "subdir\foo\bar". "\foo\bar" will only exclude "foo\bar".

```
Examples:
Source: "*"; Excludes: "*.~*"
Source: "*"; Excludes: "*.~*,\Temp\*"; Flags: rec
```

ExternalSize

This parameter must be combined with the external flag and specifies the size of the external file in bytes. If this parameter is not specified, Setup retrieves the file size at startup. Primarily useful for files that aren't available at startup, for example files located on a second disk when <u>disk spanning</u> is being used.

Example:

ExternalSize: 1048576; Flags: external

CopyMode

You should not use this parameter in any new scripts. This parameter was deprecated and replaced by flags in Inno Setup 3.0.5:

```
CopyMode: normal -> Flags: promptifolder CopyMode:
alwaysskipifsameorolder -> noflags
CopyMode: onlyifdoesntexist -> Flags:
onlyifdoesntexist
CopyMode: alwaysoverwrite -> Flags: ignoreversion
CopyMode: dontcopy -> Flags: dontcopy
```

What was CopyMode: alwaysskipifsameorolder is now the default behavior. (The previous default was CopyMode: normal.)

Attribs

Specifies additional attributes for the file. This can include one or more of the following: readonly, hidden, system. If this parameter is not specified, Setup does not assign any special attributes to the file.

Example:

Attribs: hidden system

Permissions

Specifies additional permissions to grant in the file's ACL (access control list). It is not recommended that you use this parameter if you aren't familiar with ACLs or why you would need to change them, because misusing it could negatively impact system security.

For this parameter to have an effect the file must be located on a partition that supports ACLs (such as NTFS), and the current user must be able to change the permissions on the file. In the event these conditions are not met, no error message will be displayed, and the permissions will not be set.

This parameter should *only* be used on files private to your application. Never change the ACLs on shared system files, otherwise you can open up security holes on your users' systems.

The specified permissions are set regardless of whether the file existed prior to installation.

This parameter can include one or more space separated values in the format:

```
<user or group identifier>-<access type>
```

The following access types are supported for the [Files] section:

full

Grants "Full Control" permission, which is the same as modify (see below), but additionally allows the specified user/group to take ownership of the file and change its permissions. Use sparingly; generally, modify is sufficient.

modify

Grants "Modify" permission, which allows the specified user/group to read, execute, modify, and delete the file.

readexec

Grants "Read & Execute" permission, which allows the specified user/group to read and execute the file.

Example:

Permissions: users-modify

FontInstall

Tells Setup the file is a font that needs to be installed. The value of this parameter is the name of the font as stored in the registry or WIN.INI. This must be exactly the same name as you see when you double-click the font file in Explorer. Note that Setup will automatically append " (TrueType)" to the end of the name.

If the file is not a TrueType font, you must specify the flag fontisnttruetype in the Flags parameter.

It's recommended that you use the flags onlyifdoesntexist and uninsneveruninstall when installing fonts to the {fonts} directory.

To successfully install a font, the user must be a member of the Power Users or Administrators groups.

For compatibility with 64-bit Windows, fonts should not be installed to the sys directory. Use fonts as the destination directory instead.

```
Example:
```

```
Source: "OZHANDIN.TTF"; DestDir: "{fonts}"; FontI
```

StrongAssemblyName

Specifies the strong assembly name of the file. Used by Uninstall only.

This parameter is ignored if the gacinstall flag isn't also specified.

```
Example:
StrongAssemblyName: "MyAssemblyName, Version=1.0.
```

Flags

This parameter is a set of extra options. Multiple options may be used by separating them by spaces. The following options are supported:

32bit

Causes the {sys} constant to map to the 32-bit System directory when used in the Source and DestDir parameters, the regserver and regtypelib flags to treat the file as 32bit, and the sharedfile flag to update the 32-bit SharedDLLs registry key. This is the default behavior in a <u>32bit mode</u> install.

64bit

Causes the {sys} constant to map to the 64-bit System directory when used in the Source and DestDir parameters, the regserver and regtypelib flags to treat the file as 64bit, and the sharedfile flag to update the 64-bit SharedDLLs registry key. This is the default behavior in a <u>64bit mode</u> install.

allowunsafefiles

Disables the compiler's automatic checking for <u>unsafe files</u>. It is strongly recommended that you DO NOT use this flag, unless you are absolutely sure you know what you're doing.

comparetimestamp

(Not recommended; see below)

Instructs Setup to proceed to comparing time stamps if the file being installed already exists on the user's system, and at least one of the following conditions is true:

• Neither the existing file nor the file being installed has version info.

- The ignoreversion flag is also used on the entry.
- The replacesameversion flag isn't used, and the existing file and the file being installed have the same version number (as determined by the files' version info).

If the existing file has an older time stamp than the file being installed, the existing file will replaced. Otherwise, it will not be replaced.

Use of this flag is *not recommended* except as a last resort, because there is an inherent issue with it: NTFS partitions store time stamps in UTC (unlike FAT partitions), which causes local time stamps -- what Inno Setup works with by default -- to shift whenever a user changes their system's time zone or when daylight saving time goes into or out of effect. This can create a situation where files are replaced when the user doesn't expect them to be, or not replaced when the user expects them to be.

confirmoverwrite

Always ask the user to confirm before replacing an existing file.

createallsubdirs

By default the compiler skips empty directories when it recurses subdirectories searching for the Source filename/wildcard. This flag causes these directories to be created at install time (just like if you created [Dirs] entries for them).

Must be combined with recursesubdirs.

deleteafterinstall

Instructs Setup to install the file as usual, but then delete it once the installation is completed (or aborted). This can be useful for extracting temporary data needed by a program executed in the script's [Run] section.

This flag will not cause existing files that weren't replaced during installation to be deleted.

This flag cannot be combined with the isreadme, regserver, regtypelib, restartreplace, sharedfile, or uninsneveruninstall flags.

dontcopy

Don't copy the file to the user's system during the normal file copying stage but do statically compile the file into the installation. This flag is useful if the file is handled by the [Code] section exclusively and extracted using ExtractTemporaryFile.

dontverifychecksum

Prevents Setup from verifying the file checksum after extraction. Use this flag on files you wish to modify while already compiled into Setup.

Must be combined with nocompression.

external

This flag instructs Inno Setup not to statically compile the file specified by the Source parameter into the installation files, but instead copy from an existing file on the distribution media or the user's system. See the Source parameter description for more information.

fontisnttruetype

Specify this flag if the entry is installing a *non-TrueType* font with the FontInstall parameter.

gacinstall

Install the file into the .NET Global Assembly Cache. When used in combination with sharedfile, the file will only be uninstalled when the reference count reaches zero.

To uninstall the file Uninstaller uses the strong assembly name specified by parameter StrongAssemblyName.

An exception will be raised if an attempt is made to use this flag on a system with no .NET Framework present.

ignoreversion

Don't compare version info at all; replace existing files regardless of their version number.

This flag should only be used on files private to your application, *never* on shared system files.

isreadme

File is the "README" file. Only *one* file in an installation can have this flag. When a file has this flag, the user will asked if he/she would like to view the README file after the installation has completed. If Yes is chosen, Setup will open the file, using the default program for the file type. For this reason, the README file should always end with an extension like .txt, .wri, or .doc.

Note that if Setup has to restart the user's computer (as a result of installing a file with the flag restartreplace or if the AlwaysRestart [Setup] section directive is yes), the user will not be given an option to view the README file.

nocompression

Prevents the compiler from attempting to compress the file. Use this flag on file types that you know can't benefit from compression (for example, JPEG images) to speed up the compilation process and save a few bytes in the resulting installation.

noencryption

Prevents the file from being stored encrypted. Use this flag if you have enabled encryption (using the [Setup] section directive Encryption) but want to be able to extract the file using the [Code] section support function <u>ExtractTemporaryFile</u> before the user has entered the correct password.

noregerror

When combined with either the regserver or regtypelib flags, Setup will not display any error message if the registration fails.

onlyifdestfileexists

Only install the file if a file of the same name already exists on the user's system. This flag may be useful if your installation is a patch to an existing installation, and you don't want files to be installed that the user didn't already have.

onlyifdoesntexist

Only install the file if it doesn't already exist on the user's system.

overwritereadonly

Always overwrite a read-only file. Without this flag, Setup will ask the user if an existing read-only file should be overwritten.

promptifolder

By default, when a file being installed has an older version number (or older time stamp, when the comparetimestamp flag is used) than an existing file, Setup will not replace the existing file. (See the *Remarks* section at the bottom of this topic for more details.) When this flag is used, Setup will ask the user whether the file should be replaced, with the default answer being to keep the existing file.

recursesubdirs

Instructs the compiler or Setup to also search for the Source filename/wildcard in subdirectories under the Source directory.

regserver

Register the DLL/OCX file. With this flag set, Setup will call the DllRegisterServer function exported by the DLL/OCX file, and the uninstaller will call DllUnregisterServer prior to removing the file. When used in combination with sharedfile, the DLL/OCX file will only be unregistered when the reference count reaches zero.

On a <u>64-bit mode</u> install, the file is assumed to be a 64-bit image and will be registered inside a 64-bit process. You can override this by specifying the 32bit flag.

See the *Remarks* at the bottom of this topic for more information.

regtypelib

Register the type library (.tlb). The uninstaller will unregister the type library (unless the flag uninsneveruninstall is specified). As with the regserver flag, when used in combination with sharedfile, the file will only be unregistered by the uninstaller when the reference count reaches zero.

On a <u>64-bit mode</u> install running on an x64 edition of Windows, the type library will be registered inside a 64-bit process. You can override this by specifying the 32bit flag.

Registering type libraries in 64-bit mode on Itanium editions of Windows is not supported.

See the *Remarks* at the bottom of this topic for more information.

replacesameversion

When this flag is used and the file already exists on the user's system and it has the same version number as the file being installed, Setup will compare the files and replace the existing file if their contents differ.

The default behavior (i.e. when this flag isn't used) is to not replace an existing file with the same version number.

restartreplace

When an existing file needs to be replaced, and it is in use (locked) by another running process, Setup will by default display an error message. This flag tells Setup to instead register the file to be replaced the next time the system is restarted (by calling MoveFileEx or by creating an entry in WININIT.INI). When this happens, the user will be prompted to restart their computer at the end of the installation process.

NOTE: This flag has no effect if the user does not have administrative privileges. Therefore, when using this flag, it is

recommended that you leave the <u>PrivilegesRequired</u> [Setup] section directive at the default setting of admin.

setntfscompression

Instructs Setup to enable NTFS compression on the file (even if it didn't replace the file). If it fails to set the compression state for any reason (for example, if compression is not supported by the file system), no error message will be displayed.

sharedfile

Specifies that the file is shared among multiple applications, and should only be removed at uninstall time if no other applications are using it. Most files installed to the Windows System directory should use this flag, including .OCX, .BPL, and .DPL files.

Windows' standard shared file reference-counting mechanism (located in the registry under

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\Cur is used to keep track of how many applications depend on the file. Each time the file is installed, the reference count for the file is incremented. (This happens regardless of whether the installer actually replaces the file on disk.) When an application using the file is uninstalled, the reference count is decremented. If the count reaches zero, the file is deleted (with the user's confirmation, unless the uninsnosharedfileprompt flag is also specified).

If Setup is run more than once, the reference count for the file will be incremented more than once. The uninstaller will decrement the reference count the same number of times, however, so no references are leaked (provided the <u>UninstallLogMode</u> [Setup] section directive isn't changed from its default setting of append).

When this flag is used, do not specify {syswow64} in the DestDir parameter; use {sys} instead. Even though {sys} and {syswow64} map to the same underlying directory in a

<u>32-bit mode</u> install, the path name must exactly match what every other existing installer is using; otherwise, a second reference count for the file would be created, which could result in the file being removed prematurely. If you need to install a shared file to the 32-bit System directory in a <u>64-bit</u> <u>mode</u> install, specify {sys} in the DestDir parameter and additionally include the 32bit flag.

skipifsourcedoesntexist

This flag instructs the compiler -- or Setup, if the external flag is also used -- to silently skip over the entry if the source file does not exist, instead of displaying an error message.

solidbreak

When <u>solid compression</u> is enabled, this flag instructs the compiler to finalize the current compression stream and begin a new one before compressing the file(s) matched by Source. This allows Setup to seek to the file instantly without having to decompress any preceding files first. May be useful in a large, multi-component installation if you find too much time is being spent decompressing files belonging to components that weren't selected.

sortfilesbyextension

This flag instructs the compiler to compress the found files sorted by extension before it sorts by path name. This potentially decreases the size of Setup if <u>solid compression</u> is also used.

sortfilesbyname

This flag instructs the compiler to compress the found files sorted by name before it sorts by path name. This potentially decreases the size of Setup if <u>solid compression</u> is also used. If sortfilesbyextension is also used, files are first sorted by extension.

touch

This flag causes Setup to set the time/date stamp of the installed file(s) to that which is specified by the <u>TouchDate</u> and

TouchTime [Setup] section directives.

This flag has no effect if combined with the external flag.

uninsnosharedfileprompt

When uninstalling the shared file, automatically remove the file if its reference count reaches zero instead of asking the user. Must be combined with the sharedfile flag to have an effect.

uninsremovereadonly

When uninstalling the file, remove any read-only attribute from the file before attempting to delete it.

uninsrestartdelete

When this flag is used and the file is in use at uninstall time, the uninstaller will queue the file to be deleted when the system is restarted, and at the end of the uninstallation process ask the user if he/she wants to restart. This flag can be useful when uninstalling things like shell extensions which cannot be programmatically stopped. Note that administrative privileges are required for this flag to have an effect.

uninsneveruninstall

Never remove the file. This flag can be useful when installing very common shared files that shouldn't be deleted under any circumstances, such as MFC DLLs.

Note that if this flag is combined with the sharedfile flag, the file will never be deleted at uninstall time but the reference count will still be properly decremented.

unsetntfscompression

Instructs Setup to disable NTFS compression on the file (even if it didn't replace the file). If it fails to set the compression state for any reason (for example, if compression is not supported by the file system), no error message will be displayed.

Example:

Flags: isreadme

Components and Tasks Parameters

Common Parameters

Remarks

If a file already exists on the user's system, it by default will be replaced according to the following rules:

- 1. If the existing file is an older version than the file being installed (as determined by the files' version info), the existing file will be replaced.
- 2. If the existing file is the same version as the file being installed, the existing file will not be replaced, except if the replacesameversion flag is used and the content of the two files differs.
- 3. If the existing file is a newer version than the file being installed, or if the existing file has version info but the file being installed does not, the existing file will not be replaced.
- 4. If the existing file does not have version info, it will be replaced.

Certain flags such as onlyifdoesntexist, ignoreversion, and promptifolder alter the aforementioned rules.

If Setup is unable to replace an existing file because it is in use by another process, it will make up to 4 additional attempts to replace the file, delaying one second before each attempt. If all attempts fail, an error message will be displayed.

Setup registers all files with the regserver or regtypelib flags as the last step of installation. However, if the [Setup] section directive AlwaysRestart is yes, or if there are files with the restartreplace flag, all files get registered on the next reboot (by creating an entry in Windows' *RunOnce* registry key).

When files with a .HLP extension (Windows help files) are uninstalled, the corresponding .GID and .FTS files are automatically deleted as well. Similarly, when a .CHM (HTML Help) file is deleted, any .CHW (generated index) file is automatically deleted.

[lcons] section

This optional section defines any shortcuts Setup is to create in the Start Menu and/or other locations, such as the desktop.

Here is an example of an [Icons] section:

```
[Icons]
Name: "{group}\My Program"; Filename: "{app}\MYPR
Name: "{group}\Uninstall My Program"; Filename: "
```

The following is a list of the supported parameters:

Name (Required)

The name and location of the shortcut to create. Any of the shell folder constants or directory constants may be used in this parameter.

Keep in mind that shortcuts are stored as literal files so any characters not allowed in normal filenames can't be used here. Also, because it's not possible to have two files with the same name, it's therefore not possible to have two shortcuts with the same name.

```
Examples:
Name: "{group}\My Program"
Name: "{group}\Subfolder\My Program"
Name: "{commondesktop}\My Program"
Name: "{commonprograms}\My Program"
Name: "{commonstartup}\My Program"
```

Filename (Required)

The command line filename for the shortcut, which normally begins with a directory constant.

In addition to file and folder names, URLs (web site addresses) may also be specified. When a URL is specified, Setup will create an "Internet Shortcut" (.url) file, and ignore the Parameters, WorkingDir, HotKey, and Comment parameters.

On 64-bit Windows, note that the {sys} constant will map to the native 64-bit System directory when the shortcut is launched by a 64-bit process, such as Windows Explorer. This is true regardless of whether the install is running in <u>64-bit mode</u>. To create a shortcut that always points to the 32-bit System directory, use {syswow64} instead. (The same applies to the WorkingDir and IconFilename parameters.)

Examples:	
Filename:	"{app}\MYPROG.EXE"
Filename:	"{uninstallexe}"
Filename:	"{app}\FolderName"
Filename:	"http://www.example.com/"

Parameters

Optional command line parameters for the shortcut, which can include constants.

```
Example:
```

Parameters: "/play filename.mid"

WorkingDir

The working (or *Start In*) directory for the shortcut, which specifies the initial current directory for the program. This parameter can include constants.

If this parameter is not specified or is blank, Setup will try to extract a directory name from the Filename parameter. If that fails (unlikely), the working directory will be set to {sys}.

```
Example:
WorkingDir: "{app}"
```

HotKey

The hot key (or "shortcut key") setting for the shortcut, which is a combination of keys with which the program can be started.

Note: If you change the shortcut key and reinstall the application, Windows may continue to recognize old shortcut key(s) until you log off and back on or restart the system.

Example: HotKey: "ctrl+alt+k"

Comment

Specifies the *Comment* (or "description") field of the shortcut, which determines the popup hint for it. This parameter can include constants.

Example:

Comment: "This is my program"

IconFilename

The filename of a custom icon (located on the user's system) to be displayed. This can be an executable image (.exe, .dll) containing icons or a .ico file. If this parameter is not specified or is blank, Windows will use the file's default icon. This parameter can include constants.

```
Example:
IconFilename: "{app}\myicon.ico"
```

Note: when Setup is running on 64-bit Windows, it will automatically replace {pf32}\ in the filename with '%ProgramFiles(x86)%\' to work around a bug in 64-bit Windows: 64-bit Windows replaces {pf32}\ with '%ProgramFiles%\' which is incorrect.

IconIndex

Zero-based index of the icon to use in the file specified by

IconFilename. Defaults to 0.

If IconIndex is non-zero and IconFilename is not specified or is blank, it will act as if IconFilename is the same as Filename.

```
Example:
IconIndex: 0
```

AppUserModelID

Specifies the Windows 7 Application User Model ID for the shortcut. Ignored on earlier Windows versions. This parameter can include constants.

Example:

AppUserModelID: "MyCompany.MyProg"

Flags

This parameter is a set of extra options. Multiple options may be used by separating them by spaces. The following options are supported:

closeonexit

When this flag is set, Setup will set the "Close on Exit" property of the shortcut. This flag only has an effect if the shortcut points to an MS-DOS application (if it has a .pif extension, to be specific). If neither this flag nor the dontcloseonexit flags are specified, Setup will not attempt to change the "Close on Exit" property.

createonlyiffileexists

When this flag is set, the installer will only try to create the icon if the file specified by the Filename parameter exists.

dontcloseonexit

Same as closeonexit, except it causes Setup to uncheck the "Close on Exit" property.

excludefromshowinnewinstall

Prevents the Start menu entry for the new shortcut from receiving a highlight on Windows 7 and additionally prevents the new shortcut from being automatically pinned the Start screen on Windows 8 (or later). Ignored on earlier Windows versions.

foldershortcut

Creates a special type of shortcut known as a "Folder Shortcut". Normally, when a shortcut to a folder is present on the Start Menu, clicking the item causes a separate Explorer window to open showing the target folder's contents. In contrast, a "folder shortcut" will show the contents of the target folder as a submenu instead of opening a separate window.

This flag is currently ignored when running on Windows 7 (or later), as folder shortcuts do not expand properly on the Start Menu anymore. It is not known whether this is a bug in Windows 7 or a removed feature.

When this flag is used, a folder name must be specified in the Filename parameter. Specifying the name of a file will result in a non-working shortcut.

preventpinning

Prevents a Start menu entry from being pinnable to Taskbar or the Start Menu on Windows 7 (or later). This also makes the entry ineligible for inclusion in the Start menu's Most Frequently Used (MFU) list. Ignored on earlier Windows versions.

runmaximized

When this flag is set, Setup sets the "Run" setting of the icon to "Maximized" so that the program will be initially maximized when it is started.

runminimized

When this flag is set, Setup sets the "Run" setting of the icon to "Minimized" so that the program will be initially minimized when it is started.
uninsneveruninstall

Instructs the uninstaller not to delete the icon.

useapppaths

When this flag is set, specify just a filename (no path) in the Filename parameter, and Setup will retrieve the pathname from the

"HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\Cu Paths" registry key and prepend it to the filename automatically.

Example:

Flags: runminimized

Components and Tasks Parameters

Common Parameters

[INI] section

This optional section defines any .INI file entries you would like Setup to set on the user's system.

Here is an example of an [INI] section:

```
[INI]
Filename: "MyProg.ini"; Section: "InstallSettings
Filename: "MyProg.ini"; Section: "InstallSettings
```

The following is a list of the supported parameters:

Filename (Required)

The name of the .INI file you want Setup to modify, which can include constants. If this parameter does not include a path, it will write to the Windows directory. If this parameter is blank, it will write to WIN.INI in the Windows directory.

```
Example:
Filename: "{app}\MyProg.ini"
```

Section (Required)

The name of the section in which to create the entry, which can include constants.

```
Example:
Section: "Settings"
```

Key

The name of the key to set, which can include constants. If this parameter is not specified or is blank, no key is created.

```
Example:
Key: "Version"
```

String

The value to assign to the key, which can use constants. If this parameter is not specified, no key is created.

Example: String: "1.0"

Flags

This parameter is a set of extra options. Multiple options may be used by separating them by spaces. The following options are supported:

createkeyifdoesntexist

Assign to the key *only* if the key doesn't already exist in the file. If this flag is *not* specified, the key will be set regardless of whether it already existed.

uninsdeleteentry

Delete the entry when the program is uninstalled. This can be combined with the uninsdeletesectionifempty flag.

uninsdeletesection

When the program is uninstalled, delete the entire section in which the entry is located. It obviously wouldn't be a good idea to use this on a section that is used by Windows itself (like some of the sections in WIN.INI). You should only use this on sections private to your application.

uninsdeletesectionifempty

Same as uninsdeletesection, but deletes the section only if there are no keys left in it. This can be combined with the uninsdeleteentry flag.

Example:

Flags: uninsdeleteentry

Components and Tasks Parameters

Common Parameters

[InstallDelete] section

This optional section is identical in format to the [UninstallDelete] section, except its entries are processed as the first step of *installation*.

[Languages] section

Inno Setup supports multilingual installations. The [Languages] section defines the languages to make available to the Setup program.

Setup determines the default language to use for its messages in the following order:

Non-Unicode Inno Setup:

- 1. It searches for a language whose LanguageID setting (normally specified in the [LangOptions] section of the language's .isl file) matches both the primary language identifier and sublanguage identifier of the current user's UI language or locale (depending on the setting of LanguageDetectionMethod), and whose LanguageCodePage setting matches the system code page.
- 2. If no match is found, it searches for just a primary language identifier and code page match. If two or more available languages have the same primary language identifier and code page, it selects the first one listed in the [Languages] section.
- 3. If no match is found, it defaults to the first language specified in the [Languages] section.

Unicode Inno Setup:

- 1. It searches for a language whose LanguageID setting (normally specified in the [LangOptions] section of the language's .isl file) matches both the primary language identifier and sublanguage identifier of the current user's UI language or locale (depending on the setting of LanguageDetectionMethod).
- 2. If no match is found, it searches for just a primary language identifier match. If two or more available languages have the same primary language identifier, it selects the first one listed in the [Languages] section.

Exception: Simplified Chinese is excluded from consideration in this step if the user's UI language or locale (depending on the setting of <u>LanguageDetectionMethod</u>) is Traditional Chinese, and vice versa.

3. If no match is found, it defaults to the first language specified in the [Languages] section.

If the <u>ShowLanguageDialog</u> [Setup] section directive is set to yes (the default), a *Select Language* dialog will be displayed which gives the user an opportunity to override the language Setup chose. On non Unicode Inno Setup languages that can't be displayed properly on the user's system will be hidden. See the [LangOptions] section help topic for details.

The following is an example of a [Languages] section. It defines two languages: English, based on the standard Default.isl file, and Dutch, based on a third-party translation.

```
[Languages]
Name: "en"; MessagesFile: "compiler:Default.isl"
Name: "nl"; MessagesFile: "compiler:Languages\Dut
```

Name (Required)

The internal name of the language, which you can set to anything you like. This can used as a prefix on [LangOptions] or [Messages] section entries to have the entries apply to only one language. The {language} constant returns the internal name of the selected language.

```
Example:
Name: "en"
```

MessagesFile (Required)

Specifies the name(s) of file(s) to read the default messages from. The file(s) must be located in your installation's <u>source directory</u> when running the Setup Compiler, unless a fully qualified pathname is specified or the pathname is prefixed by "compiler:", in which case it looks for the file in the Compiler directory.

When multiple files are specified, they are read in the order they are specified, thus the last message file overrides any messages in

previous files.

See the [Messages] section help topic for details on the format of .isl files.

Examples:

```
MessagesFile: "compiler:Dutch.isl"
MessagesFile: "compiler:Default.isl,compiler:MyMe
```

LicenseFile

Specifies the name of an optional license agreement file, in .txt or .rtf (rich text) format, which is displayed before the user selects the destination directory for the program. This file must be located in your installation's <u>source directory</u> when running the Setup Compiler, unless a fully qualified pathname is specified or the pathname is prefixed by "compiler:", in which case it looks for the file in the Compiler directory.

```
Example:
```

LicenseFile: "license-Dutch.txt"

InfoBeforeFile

Specifies the name of an optional "readme" file, in .txt or .rtf (rich text) format, which is displayed before the user selects the destination directory for the program. This file must be located in your installation's <u>source directory</u> when running the Setup Compiler, unless a fully qualified pathname is specified or the pathname is prefixed by "compiler:", in which case it looks for the file in the Compiler directory.

Example:

InfoBeforeFile: "infobefore-Dutch.txt"

InfoAfterFile

Specifies the name of an optional "readme" file, in .txt or .rtf (rich

text) format, which is displayed after a successful install. This file must be located in your installation's <u>source directory</u> when running the Setup Compiler, unless a fully qualified pathname is specified or the pathname is prefixed by "compiler:", in which case it looks for the file in the Compiler directory.

This differs from isreadme files in that this text is displayed as a page of the wizard, instead of in a separate Notepad window.

Example:

```
InfoAfterFile: "infoafter-Dutch.txt"
```

[Messages] section

A [Messages] section is used to define the messages displayed by the Setup program and uninstaller. Normally, you need not create a [Messages] section in your script file, since all messages are, by default, pulled in from the file *Default.isl* included with Inno Setup (or whichever file is specified by a [Languages] section entry).

However, particular messages can be overridden by creating a [Messages] section in your script file. To do this, first you will need to know the ID of the message you want to change. This can be easily found by searching Default.isl. For example, say you wanted to change the "&Next >" button on the wizard to read "&Forward >". The ID of this message is "ButtonNext", so you would create a [Messages] section like this:

[Messages] ButtonNext=&Forward >

Some messages take arguments such as %1 and %2. You can rearrange the order of the arguments (i.e. move the %2 before a %1) and also duplicate arguments if needed (i.e. "%1 ... %1 %2"). On messages with arguments, use two consecutive "%" characters to embed a single "%". "%n" creates a line break.

If you wish to translate all of Inno Setup's text to another language, instead of modifying Default.isl or overriding each message in every script you create, make a copy of Default.isl with another name like *MyTranslation.isl.* On any installation you wish to use MyTranslation.isl, create a [Languages] section entry pointing to the file.

In cases where there are multiple [Languages] section entries, specifying a [Messages] section entry in your script (as opposed to an .isl file) will by default override that message for all languages. To apply a [Messages] section entry to only one language, prefix it with the language's internal name followed by a period. For example:

```
en.ButtonNext=&Forward >
```

Special-purpose IDs

The special-purpose BeveledLabel message can be used to specify a line of text that is shown in the lower left corner of the wizard window and uninstaller window. The following is an example:

[Messages] BeveledLabel=Inno Setup

[CustomMessages] section

A [CustomMessages] section is used to define the custom message values for {cm:...} constants. See the <u>Constants</u> documentation for more information.

An example of a task with a description taken from the [CustomMessages] section using a {cm:...} constant:

```
[CustomMessages]
CreateDesktopIcon=Create a &desktop icon
[Tasks]
Name: desktopicon; Description: "{cm:CreateDeskto
```

Messages may take arguments, from %1 up to %9. You can rearrange the order of the arguments (i.e. move the %2 before a %1) and also duplicate arguments if needed (i.e. "%1 ... %1 %2"). On messages with arguments, use two consecutive "%" characters to embed a single "%". "%n" creates a line break.

In cases where there are multiple [Languages] section entries, specifying a [CustomMessages] section entry in your script (as opposed to an .isl file) will by default override that message for all languages. To apply a [CustomMessages] section entry to only one language, prefix it with the language's internal name followed by a period. For example:

```
nl.CreateDesktopIcon=Maak een snelkoppeling op he
```

Currently, the .isl files for all languages that come with Inno Setup have the following custom messages defined and translated for each language (shown here with their English values):

```
NameAndVersion=%1 version %2
AdditionalIcons=Additional icons:
CreateDesktopIcon=Create a &desktop icon
CreateQuickLaunchIcon=Create a &Quick Launch icon
```

```
ProgramOnTheWeb=%1 on the Web
UninstallProgram=Uninstall %1
LaunchProgram=Launch %1
AssocFileExtension=&Associate %1 with the %2 file
AssocingFileExtension=Associating %1 with the %2
AutoStartProgramGroupDescription=Startup:
AutoStartProgram=Automatically start %1
AddonHostProgramNotFound=%1 could not be located
```

You may use these predefined custom messages in your own script. An example which uses UninstallProgram:

```
[Icons]
Name: "{group}\{cm:UninstallProgram,My Program}";
```

[LangOptions] section

A [LangOptions] section is used to define the language-specific settings, such as fonts, used by the Setup program and uninstaller. Normally, you need not create a [LangOptions] section in your script file, since the language-specific settings are, by default, pulled in from the file *Default.isl* included with Inno Setup (or whichever file is specified by a [Languages] section entry).

The following is an example of a [LangOptions] section. (The settings listed below are the defaults.)

[LangOptions] LanguageName=English LanguageID=\$0409 LanguageCodePage=0 DialogFontName= DialogFontSize=8 WelcomeFontSize=8 WelcomeFontSize=12 TitleFontName=Arial TitleFontSize=29 CopyrightFontName=Arial CopyrightFontSize=8 RightToLeft=no

LanguageName is the native name of the language (so not the English name). It is displayed in the list of available languages on the *Select Language* dialog in a multilingual installation. It is internally stored as a Unicode string and displayed as such. To embed Unicode characters, use "<nnnn>", where "nnnn" is the 4-digit hexadecimal Unicode characters using the Character Map accessory included with Windows 2000 and later.

LanguageID is the numeric "language identifier" of the language. Refer to the <u>list of valid language identifiers on MSDN</u>. This, along with LanguageCodePage, is used for the purpose of auto-detecting the most appropriate language to use by default, so be sure it is set correctly. It should always begin with a "\$" sign, since language identifiers are in hexadecimal. If no language identifier currently exists for the language, set this to zero.

Non Unicode Inno Setup LanguageCodePage:

LanguageCodePage specifies the "code page" (character set) needed for Setup to display the language. When auto-detecting the most appropriate language to use by default, it only considers languages whose LanguageCodePage values match the system code page. In addition, when populating the list of available languages on the Select Language dialog in a multilingual installation, it likewise only considers languages whose LanguageCodePage values match the system code page (except if <u>ShowUndisplayableLanguages</u> is set). The goal of this is to hide languages that can't be displayed properly on the user's system. For example, Russian text can't be displayed properly unless the code page is 1251, so there is little reason to list Russian as an option if the system is running in a different code page. If LanguageCodePage is set to 0, the language will always be considered, regardless of the system code page. It makes sense to use 0 on languages that contain pure ASCII, such as English, since ASCII is identical across all code pages.

Unicode Inno Setup LanguageCodePage:

LanguageCodePage specifies the "code page" (character set) needed for the compiler to convert the text in the language's .isl file to Unicode. Note that text in the .iss file such as a [CustomMessages] entry for the language is not converted and should be in Unicode already. If LanguageCodePage is set to 0, the system code page will be used.

DialogFontName and **DialogFontSize** specify the font name and point size to use in dialogs. If no DialogFontName setting is present, then the value of the <u>DefaultDialogFontName</u> [Setup] section directive is used for the font name. If the specified font name does not exist on the user's system or is an empty string, 8-point *Microsoft Sans Serif* or *MS Sans Serif* will be substituted.

WelcomeFontName and **WelcomeFontSize** specify the font name and point size to use at the top of the *Welcome* and *Setup Completed* wizard pages. If the specified font name does not exist on the user's system or is an empty string, 12-point *Microsoft Sans Serif* or *MS Sans Serif* will be substituted.

TitleFontName and **TitleFontSize** specify the font name and point size to use when displaying the application name on the background window (only visible when WindowVisible=yes). If the specified font name does not exist on the user's system, 29-point *Arial* will be substituted. If the specified font name is an empty string, 29-point *Microsoft Sans Serif* or *MS Sans Serif* will be substituted.

CopyrightFontName and **CopyrightFontSize** specify the font name and point size to use when displaying the AppCopyright message on the background window (only visible when WindowVisible=yes). If the specified font name does not exist on the user's system, 8-point *Arial* will be substituted. If the specified font name is an empty string, 8point *Microsoft Sans Serif* or *MS Sans Serif* will be substituted.

RightToLeft specifies whether the language is written from right to left. If set to yes, text alignment and reading order will be reversed (with some intentional exceptions), and controls will be arranged from right to left ("flipped").

In cases where there are multiple [Languages] section entries, specifying a [LangOptions] section directive in your script (as opposed to an .isl file) will by default override that directive for all languages. To apply a [LangOptions] section directive to only one language, prefix it with the language's internal name followed by a period. For example:

en.LanguageName=English

[Registry] section

This optional section defines any registry keys/values you would like Setup to create, modify, or delete on the user's system.

By default, registry keys and values created by Setup are not deleted at uninstall time. If you want the uninstaller to delete keys or values, you must include one of the uninsdelete* flags described below.

The following is an example of a [Registry] section.

```
[Registry]
Root: HKCU; Subkey: "Software\My Company"; Flags:
Root: HKCU; Subkey: "Software\My Company\My Progr
Root: HKLM; Subkey: "Software\My Company"; Flags:
Root: HKLM; Subkey: "Software\My Company\My Progr
Root: HKLM; Subkey: "Software\My Company\My Progr
```

The following is a list of the supported parameters:

Root (Required)

The root key. This must be one of the following values:

HKCR	(HKEY_CLASSES_ROOT)
HKCU	(HKEY_CURRENT_USER)
HKLM	(HKEY_LOCAL_MACHINE)
HKU	(HKEY_USERS)
НКСС	(HKEY_CURRENT_CONFIG)

The values may have a suffix of 32 or 64. Root key values with a suffix of 32 (for example, HKLM32) map to the 32-bit view of the registry; root key values with a suffix of 64 (for example, HKLM64) map to the 64-bit view of the registry.

Root key values with a suffix of 64 can only be used when Setup is running on 64-bit Windows, otherwise an error will occur. On an installation supporting both 32- and 64-bit architectures, it is possible to avoid the error by adding a Check: IsWin64 parameter, which will cause the entry to be silently skipped when running on 32-bit Windows.

A root key value without a suffix (for example, HKLM) is equivalent to the value with a suffix of 32 (for example, HKLM32) unless the install is running in <u>64-bit mode</u>, in which case it is equivalent to the value with a suffix of 64 (for example, HKLM64).

```
Example:
Root: HKCU
```

Subkey (Required)

The subkey name, which can include constants.

Example:

Subkey: "Software\My Company\My Program"

ValueType

The data type of the value. This must be one of the following:

```
none
string
expandsz
multisz
dword
qword
binary
```

If none (the default setting) is specified, Setup will create the key but *not* a value. In this case the ValueName and ValueData parameters are ignored.

If string is specified, Setup will create a string (REG_SZ) value. If expandsz is specified, Setup will create an expand-string (REG_EXPAND_SZ) value.

If multisz is specified, Setup will create an multi-string (REG_MULTI_SZ) value.

If dword is specified, Setup will create a 32-bit integer (REG_DWORD) value.

If qword is specified, Setup will create a 64-bit integer (REG_QWORD) value.

If binary is specified, Setup will create a binary (REG_BINARY) value.

Example:

ValueType: string

ValueName

The name of the value to create, which can include constants. If this is blank, it will write to the "Default" value. If the ValueType parameter is set to none, this parameter is ignored.

Example: ValueName: "Version"

ValueData

The data for the value. If the ValueType parameter is string, expandsz, or multisz, this is a string that can include constants. If the data type is dword or qword, this can be a decimal integer (e.g. "123"), a hexadecimal integer (e.g. "\$7B"), or a constant which resolves to an integer. If the data type is binary, this is a sequence of hexadecimal bytes in the form: "00 ff 12 34". If the data type is none, this is ignored.

On a string, expandsz, or multisz type value, you may use a special constant called {olddata} in this parameter. {olddata} is replaced with the previous data of the registry value. The {olddata} constant can be useful if you need to append a string to an existing value, for example, {olddata}; {app}. If the value does not exist or the existing value isn't a string type, the {olddata} constant is silently removed. {olddata} will also be silently removed if the value being created is a multisz type but the existing value is not a multi-string type (i.e. it's REG_SZ or REG_EXPAND_SZ), and vice versa.

On a multisz type value, you may use a special constant called {break} in this parameter to embed line breaks (nulls).

Example: ValueData: "1.0"

Permissions

Specifies additional permissions to grant in the registry key's ACL (access control list). It is not recommended that you use this parameter if you aren't familiar with ACLs or why you would need to change them, because misusing it could negatively impact system security.

For this parameter to have an effect the current user must be able to change the permissions on the registry key. In the event these conditions are not met, no error message will be displayed, and the permissions will not be set.

This parameter should *only* be used on registry keys private to your application. Never change the ACLs on a top-level key like HKEY_LOCAL_MACHINE\SOFTWARE, otherwise you can open up security holes on your users' systems.

The specified permissions are set regardless of whether the registry key existed prior to installation. The permissions are not set if ValueType is none and the deletekey flag or deletevalue flag is used.

On Itanium editions of Windows, this parameter is only effectual on 32-bit registry keys. (There is no such limitation on x64 editions of Windows.)

This parameter can include one or more space separated values in the format:

```
<user or group identifier>-<access type>
```

The following access types are supported for the [Registry] section:

full

Grants "Full Control" permission, which is the same as modify (see below), but additionally allows the specified user/group to take ownership of the registry key and change its permissions. Use sparingly; generally, modify is sufficient.

modify

Grants "Modify" permission, which allows the specified user/group to read, create, modify, and delete values and subkeys.

read

Grants "Read" permission, which allows the specified user/group to read values and subkeys.

Example:

Permissions: users-modify

Flags

This parameter is a set of extra options. Multiple options may be used by separating them by spaces. The following options are supported:

createvalueifdoesntexist

When this flag is specified, Setup will create the value *only* if a value of the same name doesn't already exist. This flag has no effect if the data type is none, or if you specify the deletevalue flag.

deletekey

When this flag is specified, Setup will first try deleting the entire key if it exists, including all values and subkeys in it. If ValueType is not none, it will then create a new key and value.

To prevent disasters, this flag is ignored during installation if Subkey is blank or contains only backslashes.

deletevalue

When this flag is specified, Setup will first try deleting the value if it exists. If ValueType is not none, it will then create the key if it didn't already exist, and the new value.

dontcreatekey

When this flag is specified, Setup will not attempt to create the key or any value if the key did not already exist on the user's system. No error message is displayed if the key does not exist.

Typically this flag is used in combination with the uninsdeletekey flag, for deleting keys during uninstallation but not creating them during installation.

noerror

Don't display an error message if Setup fails to create the key or value for any reason.

preservestringtype

This is only applicable when the ValueType parameter is string or expandsz. When this flag is specified and the value did not already exist or the existing value isn't a string type (REG_SZ or REG_EXPAND_SZ), it will be created with the type specified by ValueType. If the value did exist and is a string type, it will be replaced with the same value type as the pre-existing value.

uninsclearvalue

When the program is uninstalled, set the value's data to a null string (type REG_SZ). This flag cannot be combined with the uninsdeletekey flag.

uninsdeletekey

When the program is uninstalled, delete the entire key, including all values and subkeys in it. It obviously wouldn't be a good idea to use this on a key that is used by Windows itself. You should only use this on keys private to your application.

To prevent disasters, this flag is ignored during installation if

Subkey is blank or contains only backslashes.

uninsdeletekeyifempty

When the program is uninstalled, delete the key if it has no values or subkeys left in it. This flag can be combined with uninsdeletevalue.

To prevent disasters, this flag is ignored during installation if Subkey is blank or contains only backslashes.

uninsdeletevalue

Delete the value when the program is uninstalled. This flag can be combined with uninsdeletekeyifempty.

NOTE: In Inno Setup versions prior to 1.1, you could use this flag along with the data type none and it would function as a "delete key if empty" flag. This technique is no longer supported. You must now use the uninsdeletekeyifempty flag to accomplish this.

Example:

Flags: uninsdeletevalue

Components and Tasks Parameters

Common Parameters

[Run] & [UninstallRun] sections

The [Run] section is optional, and specifies any number of programs to execute after the program has been successfully installed, but before the Setup program displays the final dialog. The [UninstallRun] section is optional as well, and specifies any number of programs to execute as the first step of *uninstallation*. Both sections share an identical syntax, except where otherwise noted below.

Programs are executed in the order they appear in the script. By default, when processing a [Run]/[UninstallRun] entry, Setup/Uninstall will wait until the program has terminated before proceeding to the next one, unless the nowait, shellexec, or waituntilidle flags are used.

Note that by default, if a program executed in the [Run] section queues files to be replaced on the next reboot (by calling MoveFileEx or by modifying wininit.ini), Setup will detect this and prompt the user to restart the computer at the end of installation. If you don't want this, set the <u>RestartIfNeededByRun</u> directive to no.

The following is an example of a [Run] section.

```
[Run]
Filename: "{app}\INIT.EXE"; Parameters: "/x"
Filename: "{app}\README.TXT"; Description: "View
Filename: "{app}\MYPROG.EXE"; Description: "Launc
```

The following is a list of the supported parameters:

Filename (Required)

The program to execute, or file/folder to open. If Filename is not an executable (.exe or .com) or batch file (.bat or .cmd), you *must* use the shellexec flag on the entry. This parameter can include constants.

```
Example:
Filename: "{app}\INIT.EXE"
```

Description

Valid only in a [Run] section. The description of the entry, which can include constants. This description is used for entries with the postinstall flag. If the description is not specified for an entry, Setup will use a default description. This description depends on the type of the entry (normal or shellexec).

Example: Description: "View the README file"

Parameters

Optional command line parameters for the program, which can include constants.

Example:

Parameters: "/x"

WorkingDir

The initial current directory for the program. If this parameter is not specified or is blank, it uses the directory from the Filename parameter; if Filename does not include a path, it will use a default directory. This parameter can include constants.

```
Example:
WorkingDir: "{app}"
```

StatusMsg

Valid only in a [Run] section. Determines the message displayed on the wizard while the program is executed. If this parameter is not specified or is blank, a default message of "Finishing installation..." will be used. This parameter can include constants.

```
Example:
StatusMsg: "Installing BDE..."
```

RunOnceld

Valid only in an [UninstallRun] section. If the same application is installed more than once, "run" entries will be duplicated in the uninstall log file. By assigning a string to RunOnceId, you can ensure that a particular [UninstallRun] entry will only be executed once during uninstallation. For example, if two or more "run" entries in the uninstall log have a RunOnceId setting of "DelService", only the latest entry with a RunOnceId setting of "DelService" will be executed; the rest will be ignored. Note that RunOnceId comparisons are case-sensitive.

```
Example:
```

RunOnceId: "DelService"

Verb

Specifies the action to be performed on the file. Must be combined with the shellexec flag. Commonly available verbs include "open" and "print". If this parameter is not specified or is blank, the default verb for the file type will be used (typically "open").

Example:

Verb: "print"

Flags

This parameter is a set of extra options. Multiple options may be used by separating them by spaces. The following options are supported:

32bit

Causes the {sys} constant to map to the 32-bit System directory when used in the Filename and WorkingDir parameters. This is the default behavior in a <u>32-bit mode</u> install.

This flag cannot be combined with the shellexec flag.

64bit

Causes the {sys} constant to map to the 64-bit System directory when used in the Filename and WorkingDir parameters. This is the default behavior in a <u>64-bit mode</u> install.

This flag can only be used when Setup is running on 64-bit Windows, otherwise an error will occur. On an installation supporting both 32- and 64-bit architectures, it is possible to avoid the error by adding a Check: IsWin64 parameter, which will cause the entry to be silently skipped when running on 32-bit Windows.

This flag cannot be combined with the shellexec flag.

hidewizard

If this flag is specified, the wizard will be hidden while the program is running.

nowait

If this flag is specified, it will not wait for the process to finish executing before proceeding to the next [Run] entry, or completing Setup. Cannot be combined with waituntilidle or waituntilterminated.

postinstall

Valid only in a [Run] section. Instructs Setup to create a checkbox on the *Setup Completed* wizard page. The user can uncheck or check this checkbox and thereby choose whether this entry should be processed or not. Previously this flag was called showcheckbox.

If Setup has to restart the user's computer (as a result of installing a file with the flag restartreplace or if the AlwaysRestart [Setup] section directive is yes), there will not be an opportunity for the checkbox to be displayed and therefore the entry will never be processed.

The isreadme flag for entries in the [Files] section is now obsolete. If the compiler detects a entry with an isreadme

flag, it strips the isreadme flag from the [Files] entry and inserts a generated [Run] entry at the head of the list of [Run] entries. This generated [Run] entry runs the README file and has flags shellexec, skipifdoesntexist, postinstall and skipifsilent.

runascurrentuser

If this flag is specified, the spawned process will inherit Setup/Uninstall's user credentials (typically, full administrative privileges).

This is the default behavior when the postinstall flag is not used.

This flag cannot be combined with the runasoriginaluser flag.

runasoriginaluser

Valid only in a [Run] section. If this flag is specified and the system is running Windows Vista or later, the spawned process will execute with the (normally non-elevated) credentials of the user that started Setup initially (i.e., the "pre-UAC dialog" credentials).

This is the default behavior when the postinstall flag is used.

If a user launches Setup by right-clicking its EXE file and selecting "Run as administrator", then this flag, unfortunately, will have no effect, because Setup has no opportunity to run any code with the original user credentials. The same is true if Setup is launched from an already-elevated process. Note, however, that this is not an Inno Setup-specific limitation; Windows Installer-based installers cannot return to the original user credentials either in such cases.

This flag cannot be combined with the runascurrentuser flag.

runhidden

If this flag is specified, it will launch the program in a hidden window. Never use this flag when executing a program that may prompt for user input.

runmaximized

If this flag is specified, it will launch the program or document in a maximized window.

runminimized

If this flag is specified, it will launch the program or document in a minimized window.

shellexec

This flag is required if Filename is not a directly executable file (an .exe or .com file). When this flag is set, Filename can be a folder or any registered file type -- including .chm, .doc, and so on. The file will be opened with the application associated with the file type on the user's system, the same way it would be if the user double-clicked the file in Explorer.

By default, when the shellexec flag is used it will not wait until the spawned process terminates. If you need that, you must add the flag waituntilterminated. Note that it cannot and will not wait if a new process isn't spawned -- for example, if Filename specifies a folder.

skipifdoesntexist

If this flag is specified in the [Run] section, Setup won't display an error message if Filename doesn't exist.

If this flag is specified in the [UninstallRun] section, the uninstaller won't display the "some elements could not be removed" warning if Filename doesn't exist.

When this flag is used, Filename must be an absolute path.

skipifnotsilent

Valid only in a [Run] section. Instructs Setup to skip this entry if Setup is not running (very) silent.

skipifsilent

Valid only in a [Run] section. Instructs Setup to skip this entry if Setup is running (very) silent.

unchecked

Valid only in a [Run] section. Instructs Setup to initially uncheck the checkbox. The user can still check the checkbox if he/she wishes to process the entry. This flag is ignored if the postinstall flag isn't also specified.

waituntilidle

If this flag is specified, it will wait until the process is waiting for user input with no input pending, instead of waiting for the process to terminate. (This calls the *WaitForInputIdle* Win32 function.) Cannot be combined with nowait or waituntilterminated.

waituntilterminated

If this flag is specified, it will wait until the process has completely terminated. Note that this is the default behavior (i.e. you don't need to specify this flag) unless you're using shellexec flag, in which case you do need to specify this flag if you want it to wait. Cannot be combined with nowait or waituntilidle.

Example:

Flags: postinstall nowait skipifsilent

Components and Tasks Parameters

Common Parameters

[UninstallDelete] section

This optional section defines any additional files or directories you want the uninstaller to delete, besides those that were installed/created using [Files] or [Dirs] section entries. Deleting .INI files created by your application is one common use for this section. The uninstaller processes these entries as the last step of uninstallation.

Here is a example of a [UninstallDelete] section:

```
[UninstallDelete]
Type: files; Name: "{win}\MYPROG.INI"
```

The following is a list of the supported parameters:

Type (Required)

Specifies what is to be deleted by the uninstaller. This must be one of the following:

files

The Name parameter specifies a name of a particular file, or a filename with wildcards.

filesandordirs

Functions the same as files except it matches directory names also, and any directories matching the name are deleted including all files and subdirectories in them.

dirifempty

When this is used, the Name parameter must be the name of a directory, but it cannot include wildcards. The directory will only be deleted if it contains no files or subdirectories.

Example:

Type: files

Name (Required)

Name of the file or directory to delete.

NOTE: Don't be tempted to use a wildcard here to delete all files in the {app} directory. Doing this is strongly recommend against for two reasons. First, users usually don't appreciate having their data files they put in the application directory deleted without warning (they might only be uninstalling it because they want to move it to a different drive, for example). It's better to leave it up to the end users to manually remove them if they want. Also, if the user happened to install the program in the wrong directory by mistake (for example, C:\WINDOWS) and then went to uninstall it there could be disastrous consequences. So again, **DON'T DO THIS!**

Example:

Name: "{win}\MYPROG.INI"

Components and Tasks Parameters

Common Parameters

Pascal Scripting: Introduction

The Pascal scripting feature (modern Delphi-like Pascal) adds lots of new possibilities to customize your Setup or Uninstall at run-time. Some examples:

- Support for aborting Setup or Uninstall startup under custom conditions.
- Support for adding custom wizard pages to Setup at run-time.
- Support for extracting and calling DLL or other files from the Pascal script before, during or after the installation.
- Support for scripted constants that can do anything the normal constants, the read-from-registry, read-from-ini and read-from-commandline constants can do + more.
- Support for run-time removal of types, components and/or tasks under custom conditions.
- Support for conditional installation of [Files], [Registry], [Run] etc. entries based on custom conditions.
- Lots of support functions to do from the Pascal script just about everything Inno Setup itself does/can do + more.

An integrated run-time debugger to debug your custom Pascal script is also available.

The scripting engine used by Inno Setup is RemObjects Pascal Script by Carlo Kok. Like Inno Setup, RemObjects Pascal Script is freely available and comes with source. See <u>http://www.remobjects.com/ps</u> for more information.

Note: the Pascal scripting feature works exclusively at run-time, and has no compile-time functionality.

See also: <u>Creating the [Code] section</u> <u>Event Functions</u> <u>Scripted Constants</u> <u>Check Parameters</u> <u>BeforeInstall and AfterInstall Parameters</u> <u>Uninstall Code</u> <u>Examples</u> <u>Support Functions Reference</u> <u>Support Classes Reference</u> <u>Using Custom Wizard Pages</u> <u>Using DLLs</u> <u>Using COM Automation objects</u>

Pascal Scripting: Creating the [Code] Section

The [Code] section is an optional section that specifies a Pascal script. A Pascal script can be used to customize Setup or Uninstall in many ways. Note that creating a Pascal script is not easy and requires experience with Inno Setup and knowledge about programming in Pascal or at least a similar programming language.

The "Code*.iss" and "UninstallCode*.iss" files in the "Examples" subdirectory in your Inno Setup directory contain various example [Code] sections. Please study them carefully before trying to create your own Pascal script.

Note: to learn more the Pascal programming language you may find useful to refer to Marco Cantu's free Essential Pascal book. See <u>http://www.marcocantu.com/epascal/</u>

Pascal Scripting: Event Functions

The Pascal script can contain several event functions which are called at appropriate times.
Setup event functions

Setup supports following event functions:

```
function InitializeSetup(): Boolean;
Called during Setup's initialization. Return False to abort Setup,
True otherwise.
```

procedure InitializeWizard();

Use this event function to make changes to the wizard or wizard pages at startup. You can't use the InitializeSetup event function for this since at the time it is triggered, the wizard form does not yet exist.

```
procedure DeinitializeSetup();
```

Called just before Setup terminates. Note that this function is called even if the user exits Setup before anything is installed.

```
procedure CurStepChanged(CurStep: <u>TSetupStep</u>);
You can use this event function to perform your own pre-install and
post-install tasks.
```

Called with CurStep=ssInstall just before the actual installation starts, with CurStep=ssPostInstall just after the actual installation finishes, and with CurStep=ssDone just before Setup terminates after a successful install.

procedure CurInstallProgressChanged(CurProgress, MaxProgress: Integer);

You can use this event function to monitor progress while Setup is extracting files, creating shortcuts, creating INI entries, and creating registry entries.

function NextButtonClick(CurPageID: Integer):
Boolean;

Called when the user clicks the Next button. If you return True, the wizard will move to the next page; if you return False, it will remain on the current page (specified by CurPageID).

Note that this function is called on silent installs as well, even

though there is no Next button that the user can click. Setup instead simulates "clicks" on the Next button. On a silent install, if your NextButtonClick function returns False prior to installation starting, Setup will exit automatically.

function BackButtonClick(CurPageID: Integer):
Boolean;

Called when the user clicks the Back button. If you return True, the wizard will move to the previous page; if you return False, it will remain on the current page (specified by CurPageID).

procedure CancelButtonClick(CurPageID: Integer; var Cancel, Confirm: Boolean);

Called when the user clicks the Cancel button or clicks the window's Close button. The Cancel parameter specifies whether normal cancel processing should occur; it defaults to True. The Confirm parameter specifies whether an "Exit Setup?" message box should be displayed; it usually defaults to True. If Cancel is set to False, then the value of Confirm is ignored.

function ShouldSkipPage(PageID: Integer): Boolean; The wizard calls this event function to determine whether or not a particular page (specified by PageID) should be shown at all. If you return True, the page will be skipped; if you return False, the page may be shown.

Note: This event function isn't called for the wpPreparing, and wpInstalling pages, nor for pages that Setup has already determined should be skipped (for example, wpSelectComponents in an install containing no components).

procedure CurPageChanged(<u>CurPageID</u>: Integer); Called after a new wizard page (specified by CurPageID) is shown.

function CheckPassword(Password: String): Boolean; If Setup finds the CheckPassword event function in the Pascal script, it automatically displays the Password page and calls CheckPassword to check passwords. Return True to accept the password and False to reject it. To avoid storing the actual password inside the compiled [Code] section which is stored inside Setup, you should use comparisons by hash only: calculate the SHA-1 hash of your salted password yourself and then compare that to

<u>GetSHA10fString</u>(Password). This way the actual value of the password remains protected.

Note: If Setup is run with a /PASSWORD= <u>command line</u> <u>parameter</u>, your CheckPassword function will be called *before* any other event function is called, including <u>InitializeSetup</u>.

function NeedRestart(): Boolean;

Return True to instruct Setup to prompt the user to restart the system at the end of a successful installation, False otherwise.

function UpdateReadyMemo(Space, NewLine, MemoUserInfoInfo, MemoDirInfo, MemoTypeInfo, MemoComponentsInfo, MemoGroupInfo, MemoTasksInfo: String): String;

If Setup finds the UpdateReadyMemo event function in the Pascal script, it is called automatically when the *Ready to Install* wizard page becomes the active page. It should return the text to be displayed in the settings memo on the *Ready to Install* wizard page as a single string with lines separated by the NewLine parameter. Parameter Space contains a string with spaces. Setup uses this string to indent settings. The other parameters contain the (possibly empty) strings that Setup would have used as the setting sections. The MemoDirInfo parameter for example contains the string for the *Selected Directory* section.

procedure RegisterPreviousData(PreviousDataKey: Integer);

To store user settings entered on custom wizard pages, place a RegisterPreviousData event function in the Pascal script and call <u>SetPreviousData(PreviousDataKey, ...)</u> inside it, once per setting.

function CheckSerial(Serial: String): Boolean;
 If Setup finds the CheckSerial event function in the Pascal

script, a serial number field will automatically appear on the User Info wizard page (which must be enabled using UserInfoPage=yes in your [Setup] section!). Return True to accept the serial number and False to reject it. When using serial numbers, it's important to keep in mind that since no encryption is used and the source code to Inno Setup is freely available, it would not be too difficult for an experienced individual to remove the serial number protection from an installation. Use this only as a convenience to the end user and double check the entered serial number (stored in the {userinfoserial} constant) in your application.

function GetCustomSetupExitCode: Integer;

Return a non zero number to instruct Setup to return a custom exit code. This function is only called if Setup was successfully run to completion and the exit code would have been 0. Also see <u>Setup</u> <u>Exit Codes</u>.

function PrepareToInstall(var NeedsRestart: Boolean): String;

You can use this event function to detect and install missing prerequisites and/or to shutdown any application which is about to be updated.

Return a non empty string to instruct Setup to stop at the <u>Preparing</u> to Install wizard page, showing the returned string as the error message. Set NeedsRestart to True if a restart is needed. This function is only called if Setup didn't already determine it can't continue because one or more files specified in the [Files] and [InstallDelete] sections were queued (by some other installation) to be replaced or deleted on the next restart.

This event function is called before Setup checks for files being inuse if <u>CloseApplications</u> is set to yes.

procedure RegisterExtraCloseApplicationsResources; To register extra files which Setup should check for being in-use if <u>CloseApplications</u> is set to yes, place a RegisterExtraCloseApplicationsResources event function in the Pascal script and call <u>RegisterExtraCloseApplicationsResource</u> inside it, once per file.

Uninstall event functions

Uninstall supports following event functions:

```
function InitializeUninstall(): Boolean;
    Return False to abort Uninstall, True otherwise.
```

```
procedure InitializeUninstallProgressForm();
```

Use this event function to make changes to the progress form at startup. You can't use the InitializeUninstall event function for this since at the time it is triggered, the progress form does not yet exist.

```
procedure DeinitializeUninstall();
```

```
procedure CurUninstallStepChanged(CurUninstallStep:
TUninstallStep);
```

function UninstallNeedRestart(): Boolean;

Return True to instruct Uninstall to prompt the user to restart the system at the end of a successful uninstallation, False otherwise.

Constants

Here's the list of constants used by these functions:

- TSetupStep values ssInstall, ssPostInstall, ssDone
- *TUninstallStep values* usAppMutexCheck, usUninstall, usPostUninstall, usDone
- PageID values for predefined <u>wizard pages</u> wpWelcome, wpLicense, wpPassword, wpInfoBefore, wpUserInfo, wpSelectDir, wpSelectComponents, wpSelectProgramGroup, wpSelectTasks, wpReady, wpPreparing, wpInstalling, wpInfoAfter, wpFinished

None of these functions are required to be present in a Pascal script.

Pascal Scripting: Scripted Constants

The Pascal script can contain several functions which are called when Setup wants to know the value of a scripted {code:...} constant. The called function must have 1 String parameter named Param, and must return a String or a Boolean value depending on where the constant is used.

The syntax of a {code:...} constant is: {code:*FunctionName*|*Param*}

- *FunctionName* specifies the name of the Pascal script function.
- *Param* specifies the string parameter to pass to the function. If you omit *Param*, an empty string will be passed.
- If you wish to include a comma, vertical bar ("|"), or closing brace ("}") inside the constant, you must escape it via "%-encoding." Replace the character with a "%" character, followed by its twodigit hex code. A comma is "%2c", a vertical bar is "%7c", and a closing brace is "%7d". If you want to include an actual "%" character, use "%25".
- *Param* may include constants. Note that you do *not* need to escape the closing brace of a constant as described above; that is only necessary when the closing brace is used elsewhere.

```
Example:
```

```
DefaultDirName={code:MyConst}\My Program
```

Here is an example of a [Code] section containing the MyConst function used above.

```
[Code]
function MyConst(Param: String): String;
begin
   Result := ExpandConstant('{pf}');
end;
```

If the function specified by the $\{code: ...\}$ constant is not included in

the [Code] section, it must be a <u>support function</u>. Here is an example.

[INI]
FileName: "{app}\MyIni.ini"; Section: "MySettings
]

See also: Constants

Pascal Scripting: Check Parameters

There is one optional <u>parameter</u> that is supported by all sections whose entries are separated into parameters. This is:

Check

The name of a check function that determines whether an entry has to be processed or not. The function must either be a custom function in the [Code] section or a <u>support function</u>.

Besides a single name, you may also use boolean expressions. See <u>Components and Tasks parameters</u> for examples of boolean expressions.

For each check function, may include a comma separated list of parameters that Setup should pass to the check function. Allowed parameter types are String, Integer and Boolean. String parameters may include constants. These constants will not be automatically expanded. If you want to pass an expanded constant, there's one special <u>support function</u> that may be called from within a parameter list for this: ExpandConstant.

Example:	
[Files]	
Source:	"MYPROG.EXE"; DestDir: "{app}"; Check: My
Source:	"A\MYFILE.TXT"; DestDir: "{app}"; Check:
Source:	"B\MYFILE.TXT"; DestDir: "{app}"; Check:
4	Let a let

All check functions must have a Boolean return value. If a check function (or the boolean expression) returns True, the entry is processed otherwise it's skipped.

Setup might call each check function several times, even if there's only one entry that uses the check function. If your function performs a lengthy piece of code, you can optimize it by performing the code only once and 'caching' the result in a global variable.

A check function isn't called if Setup already determined the entry it

shouldn't be processed.

A check function for a [Files] section entry using a wildcard is called once per file matching the wildcard. Use CurrentFileName to check for which file the function is called.

Here is an example of a [Code] section containing the check functions used above. Function DirExists is a <u>support function</u> and therefore not included in this [Code] section.

```
[Code]
var
 MyProgChecked: Boolean;
 MyProgCheckResult: Boolean;
function MyProgCheck(): Boolean;
begin
 if not MyProgChecked then begin
    MyProgCheckResult := MsgBox('Do you want to i
    MyProgChecked := True;
 end;
 Result := MyProgCheckResult;
end;
function MyDirCheck(DirName: String): Boolean;
begin
 Result := DirExists(DirName);
end;
4
                                                Þ
```

Pascal Scripting: BeforeInstall and AfterInstall Parameters

There are two optional <u>parameters</u> that are supported by all sections whose entries are separated into parameters except for [Languages], [Types], [Components] and [Tasks]. These are:

BeforeInstall

The name of a function that is to be called once just before an entry is installed. The function must either be a custom function in the [Code] section or a <u>support function</u>.

May include a comma separated list of parameters that Setup should pass to the function. Allowed parameter types are String, Integer and Boolean. String parameters may include constants. These constants will not be automatically expanded. If you want to pass an expanded constant, there's one special <u>support function</u> that may be called from within a parameter list for this: ExpandConstant.

```
Example:

[Files]

Source: "MYPROG.EXE"; DestDir: "{app}"; BeforeIns

Source: "A\MYFILE.TXT"; DestDir: "{app}"; BeforeI

Source: "B\MYFILE.TXT"; DestDir: "{app}"; BeforeI

Source: "MYPROG.CHM"; DestDir: "{app}"; BeforeIns
```

AfterInstall

The name of a function that is to be called once just after an entry is installed. The function must either be a custom function in the [Code] section or a <u>support function</u>.

May include a comma separated list of parameters that Setup should pass to the function. Allowed parameter types are String, Integer and Boolean. String parameters may include constants. These constants will not be automatically expanded. If you want to pass an expanded constant, there's one special <u>support function</u> that may be called from within a parameter list for this: ExpandConstant.

```
Example:

[Files]

Source: "MYPROG.EXE"; DestDir: "{app}"; AfterInst

Source: "A\MYFILE.TXT"; DestDir: "{app}"; AfterIn

Source: "B\MYFILE.TXT"; DestDir: "{app}"; AfterInst

Source: "MYPROG.CHM"; DestDir: "{app}"; AfterInst
```

All BeforeInstall and AfterInstall functions must not have a return value.

A BeforeInstall or AfterInstall function isn't called if Setup already determined the entry it shouldn't be processed.

A BeforeInstall or AfterInstall function for a [Files] section entry using a wildcard is called once per file matching the wildcard. Use CurrentFileName to check for which file the function is called.

Here is an example of a [Code] section containing the functions used above. Functions CurrentFileName and Log are <u>support functions</u> and therefore not included in this [Code] section.

```
[Code]
procedure MyBeforeInstall();
begin
   MsgBox('About to install MyProg.exe as ' + Curr
end;
procedure MyBeforeInstall2(FileName: String);
begin
   MsgBox('About to install ' + FileName + ' as '
end;
procedure MyAfterInstall();
begin
   MsgBox('Just installed MyProg.exe as ' + Curren
```

```
end;
procedure MyAfterInstall2(FileName: String);
begin
   MsgBox('Just installed ' + FileName + ' as ' +
end;
```

Pascal Scripting: Uninstall Code

The Pascal script can also contain code invoked at uninstall time. See the <u>Event Functions</u> topic for more information.

There is one thing that's important to be aware of when designing code to be executed at uninstall time: In cases where multiple versions of an application are installed over each other, only *one* Pascal script is run at uninstall time. Ordinarily, the script from the most recent install will be chosen. If, however, you were to *downgrade* your version of Inno Setup in a new version of your application, the script from the install built with the most recent Inno Setup version may be chosen instead. A similar situation can occur if a user installs an older version of your application over a newer one.

When producing an installation that is a "patch" for another install, and the patch install shares the same uninstall log as the original install (i.e. Uninstallable is set to yes and AppId is the set the same as the original install), make sure the patch includes a copy of the full [Code] section from the original install. Otherwise, no code would be run at uninstall time.

If, however, the patch install has Uninstallable set to no then Setup will not touch the existing uninstaller EXE or uninstall log; in this case, the patch install need not contain a copy of the [Code] section from the original install.

Pascal Scripting: Examples

The Pascal Scripting example scripts are located in separate files. Open one of the "Code*.iss" or "UninstallCode*.iss" files in the "Examples" subdirectory in your Inno Setup directory.

Pascal Scripting: Support Functions Reference

The Pascal script can call several built-in support functions.

Support functions

Here's the list of support functions that can be called from within the Pascal script.

```
Setup or Uninstall Info functions
function GetCmdTail: String;
function ParamCount: Integer;
function ParamStr(Index: Integer): String;
function <u>ActiveLanguage</u>: String;
function <u>CustomMessage(const MsgName: String):</u>
String;
function FmtMessage(const S: String; const Args:
array of String): String;
function <u>SetupMessage(const ID: TSetupMessageID):</u>
String;
function WizardDirValue: String;
function <u>WizardGroupValue</u>: String;
function <u>WizardNoIcons</u>: Boolean;
function <u>WizardSetupType(const Description:</u>
Boolean): String;
function WizardSelectedComponents(const
Descriptions: Boolean): String;
function WizardSelectedTasks(const Descriptions:
Boolean): String;
function <u>WizardSilent</u>: Boolean;
function IsUninstaller: Boolean;
function UninstallSilent: Boolean;
function <u>CurrentFilename</u>: String;
function <u>CurrentSourceFilename</u>: String;
function ExpandConstant(const S: String): String;
```

```
function ExpandConstantEx(const S: String; const
CustomConst, CustomValue: String): String;
```

```
function IsComponentSelected(const Components:
String): Boolean;
function IsTaskSelected(const Tasks: String):
Boolean;
```

```
procedure ExtractTemporaryFile(const FileName:
String);
function ExtractTemporaryFiles(const Pattern:
String): Integer;
```

```
function GetPreviousData(const ValueName,
DefaultValueData: String): String;
function SetPreviousData(const PreviousDataKey:
Integer; const ValueName, ValueData: String):
Boolean;
```

```
function Terminated: Boolean;
```

```
function
<u>RegisterExtraCloseApplicationsResource(const</u>
DisableFsRedir: Boolean; const AFilename: String):
Boolean;
function <u>RmSessionStarted</u>: Boolean;
```

Exception functions

```
procedure Abort;
procedure RaiseException(const Msg: String);
```

```
function GetExceptionMessage: String;
procedure ShowExceptionMessage;
```

System functions

function IsAdminLoggedOn: Boolean;

function IsPowerUserLoggedOn: Boolean; function GetWindowsVersion: Cardinal; procedure GetWindowsVersionEx(var Version: TWindowsVersion); function GetWindowsVersionString: String;

function IsWin64: Boolean; function Is64BitInstallMode: Boolean; function ProcessorArchitecture: TSetupProcessorArchitecture;

function InstallOnThisVersion(const MinVersion, OnlyBelowVersion: String): Boolean;

function GetEnv(const EnvVar: String): String; function GetUserNameString: String; function GetComputerNameString: String;

function GetUILanguage: Integer;

function FontExists(const FaceName: String):
Boolean;

function FindWindowByClassName(const ClassName: String): HWND; function FindWindowByWindowName(const WindowName: String): HWND; function SendMessage(const Wnd: HWND; const Msg, WParam, LParam: Longint): Longint; function PostMessage(const Wnd: HWND; const Msg, WParam, LParam: Longint): Boolean; function SendNotifyMessage(const Wnd: HWND; const Msg, WParam, LParam: Longint): Boolean; function RegisterWindowMessage(const Name: String): Longint; function SendBroadcastMessage(const Msg, WParam, LParam: Longint): Longint;

```
function PostBroadcastMessage(const Msg, WParam,
LParam: Longint): Boolean;
function SendBroadcastNotifyMessage(const Msg,
WParam, LParam: Longint): Boolean;
```

```
procedure <u>CreateMutex(const Name: String);</u>
function <u>CheckForMutexes(Mutexes: String): Boolean;</u>
```

procedure MakePendingFileRenameOperationsChecksum:
String;

```
procedure UnloadDLL(Filename: String);
function DLLGetLastError(): Longint;
```

String functions

```
function Chr(B: Byte): Char;
function Ord(C: Char): Byte;
function Copy(S: String; Index, Count: Integer):
String;
function Length(s: String): Longint;
function Lowercase(S: String): String;
function Uppercase(S: String): String;
function AnsiLowercase(S: String): String;
function AnsiUppercase(S: String): String;
function <u>StringOfChar(c: Char; I : Longint): String;</u>
procedure Delete(var S: String; Index, Count:
Integer);
procedure Insert(Source: String; var Dest: String;
Index: Integer);
function StringChange(var S: String; const FromStr,
ToStr: String): Integer;
function <u>StringChangeEx(var S: String; const</u>
FromStr, ToStr: String; const SupportDBCS: Boolean):
Integer;
function Pos(SubStr, S: String): Integer;
function AddQuotes(const S: String): String;
function <u>RemoveOuotes(const S: String): String;</u>
```

function <u>ConvertPercentStr</u>(var S: String): Boolean; function <u>CompareText(const S1, S2: string): Integer;</u> function <u>CompareStr(const S1, S2: string): Integer;</u> function Format(const Format: string; const Args: array of const): string; function Trim(const S: string): String; function TrimLeft(const S: string): String; function TrimRight(const S: string): String; function <u>StrToIntDef(s: string; def: Longint):</u> Longint; function <u>StrToInt(s: string): Longint;</u> function StrToInt64Def(s: string; def: Int64): Int64; function <u>StrToInt64(s: string): Int64;</u> function <u>StrToFloat(s: string): Extended;</u> function IntToStr(i: Int64): String; function FloatToStr(e: extended): String; function <u>CharLength(const S: String; const Index:</u> Integer): Integer; function AddBackslash(const S: String): String; function <u>RemoveBackslashUnlessRoot(const S: String):</u> String; function <u>RemoveBackslash(const S: String): String;</u> function AddPeriod(const S: String): String; function <u>ChangeFileExt(const FileName, Extension:</u> string): String; function ExtractFileExt(const FileName: string): String; function ExtractFileDir(const FileName: string): String; function ExtractFilePath(const FileName: string):

String;

function ExtractFileName(const FileName: string): String; function ExtractFileDrive(const FileName: string): String; function ExtractRelativePath(const BaseName, DestName: String): String; function ExpandFileName(const FileName: string): String; function ExpandUNCFileName(const FileName: string): String; function <u>GetDateTimeString</u>(const DateTimeFormat: String; const DateSeparator, TimeSeparator: Char): String; procedure <u>SetLength(var S: String; L: Longint);</u> procedure CharToOemBuff(var S: AnsiString); procedure OemToCharBuff(var S: AnsiString); function <u>GetMD50fString(const S: AnsiString):</u> String; function GetMD50fUnicodeString(const S: String): String; function GetSHA10fString(const S: AnsiString): String; function GetSHA10fUnicodeString(const S: String): String; function SysErrorMessage(ErrorCode: Integer):

String;

function <u>MinimizePathName(const Filename: String;</u> const Font: TFont; MaxLen: Integer): String;

Array functions

function <u>GetArrayLength(var Arr: Array): Longint;</u>

procedure <u>SetArrayLength(var Arr: Array; I:</u> Longint);

Variant functions

function <u>Null</u>: Variant; function <u>Unassigned</u>: Variant;

```
function VarIsEmpty(const V: Variant): Boolean;
function VarIsClear(const V: Variant): Boolean;
function VarIsNull(const V: Variant): Boolean;
function VarType(const V: Variant): TVarType;
```

File System functions

```
function <u>DirExists(const Name: String): Boolean;</u>
function FileExists(const Name: String): Boolean;
function FileOrDirExists(const Name: String):
Boolean;
function FileSize(const Name: String; var Size:
Integer): Boolean;
function GetSpaceOnDisk(const Path: String; const
InMegabytes: Boolean; var Free, Total: Cardinal):
Boolean;
function GetSpaceOnDisk64(const Path: String; var
Free, Total: Int64): Boolean;
function FileSearch(const Name, DirList: string):
String;
function FindFirst(const FileName: String; var
FindRec: TFindRec): Boolean;
function FindNext(var FindRec: TFindRec): Boolean;
procedure FindClose(var FindRec: TFindRec);
function <u>GetCurrentDir</u>: String;
function SetCurrentDir(const Dir: string): Boolean;
function GetWinDir: String;
function GetSystemDir: String;
```

```
function GetSysWow64Dir: String;
function GetTempDir: String;
function GetShellFolder(Common: Boolean; const ID:
```

TShellFolderID): String;

function <u>GetShellFolderByCSIDL(const Folder:</u>

```
Integer; const Create: Boolean): String;
```

function <u>GetShortName(const LongName: String):</u>
String;

```
function GenerateUniqueName(Path: String; const
Extension: String): String;
```

function GetVersionNumbers(const Filename: String; var VersionMS, VersionLS: Cardinal): Boolean; function GetVersionNumbersString(const Filename: String; var Version: String): Boolean;

```
function IsProtectedSystemFile(const Filename:
String): Boolean;
```

function GetMD50fFile(const Filename: String):
String;
function GetSHA10fFile(const Filename: String):
String;

function EnableFsRedirection(const Enable: Boolean):
Boolean;

File functions

function Exec(const Filename, Params, WorkingDir: String; const ShowCmd: Integer; const Wait: TExecWait; var ResultCode: Integer): Boolean; function ExecAsOriginalUser(const Filename, Params, WorkingDir: String; const ShowCmd: Integer; const Wait: TExecWait; var ResultCode: Integer): Boolean; function ShellExec(const Verb, Filename, Params, WorkingDir: String; const ShowCmd: Integer; const Wait: TExecWait; var ErrorCode: Integer): Boolean; function <u>ShellExecAsOriginalUser(const Verb,</u> Filename, Params, WorkingDir: String; const ShowCmd: Integer; const Wait: TExecWait; var ErrorCode: Integer): Boolean;

function <u>RenameFile(const OldName, NewName: string):</u>
Boolean;

function FileCopy(const ExistingFile, NewFile: String; const FailIfExists: Boolean): Boolean; function DeleteFile(const FileName: string):

Boolean;

procedure DelayDeleteFile(const Filename: String; const Tries: Integer);

function <u>SetNTFSCompression(const FileOrDir: String;</u> Compress: Boolean): Boolean;

function LoadStringFromFile(const FileName: String; var S: AnsiString): Boolean; function LoadStringsFromFile(const FileName: String; var S: TArrayOfString): Boolean; function SaveStringToFile(const FileName: String; const S: AnsiString; const Append: Boolean): Boolean; function SaveStringsToFile(const FileName: String; const S: TArrayOfString; const Append: Boolean): Boolean; function SaveStringsToUTF8File(const FileName: String; const S: TArrayOfString; const Append: Boolean): Boolean; function SaveStringsToUTF8File(const FileName: String; const S: TArrayOfString; const Append: Boolean):

```
function CreateDir(const Dir: string): Boolean;
function ForceDirectories(Dir: string): Boolean;
function RemoveDir(const Dir: string): Boolean;
function DelTree(const Path: String; const IsDir,
DeleteFiles, DeleteSubdirsAlso: Boolean): Boolean;
```

function CreateShellLink(const Filename, Description, ShortcutTo, Parameters, WorkingDir, IconFilename: String; const IconIndex, ShowCmd: Integer): String; function UnpinShellLink(const Filename: String): Boolean;

procedure RegisterServer(const Is64Bit: Boolean; const Filename: String; const FailCriticalErrors: Boolean); function UnregisterServer(const Is64Bit: Boolean; const Filename: String; const FailCriticalErrors: Boolean): Boolean; procedure RegisterTypeLibrary(const Is64Bit: Boolean; const Filename: String); function UnregisterTypeLibrary(const Is64Bit: Boolean; const Filename: String): Boolean procedure IncrementSharedCount(const Is64Bit: Boolean; const Filename: String; const AlreadyExisted: Boolean); function DecrementSharedCount(const Is64Bit: Boolean; const Filename: String): Boolean; procedure RestartReplace(const TempFile, DestFile: String); procedure <u>UnregisterFont</u>(const FontName, FontFilename: String); function ModifyPifFile(const Filename: String; const CloseOnExit: Boolean): Boolean;

Registry functions

function <u>RegKeyExists</u>(const RootKey: Integer; const SubKeyName: String): Boolean; function <u>RegValueExists</u>(const RootKey: Integer; const SubKeyName, ValueName: String): Boolean;

function <u>RegGetSubkeyNames(const RootKey: Integer;</u> const SubKeyName: String; var Names: TArrayOfString): Boolean; function <u>RegGetValueNames</u>(const RootKey: Integer; const SubKeyName: String; var Names: TArrayOfString): Boolean;

function <u>RegQueryStringValue(const RootKey: Integer;</u> const SubKeyName, ValueName: String; var ResultStr: String): Boolean;

function <u>RegQueryMultiStringValue(const RootKey:</u>
Integer; const SubKeyName, ValueName: String; var
ResultStr: String): Boolean;

function <u>RegQueryDWordValue(const RootKey: Integer;</u> const SubKeyName, ValueName: String; var ResultDWord: Cardinal): Boolean;

function <u>RegQueryBinaryValue(const RootKey: Integer;</u> const SubKeyName, ValueName: String; var ResultStr: AnsiString): Boolean;

function RegWriteStringValue(const RootKey: Integer; const SubKeyName, ValueName, Data: String): Boolean; function RegWriteExpandStringValue(const RootKey: Integer; const SubKeyName, ValueName, Data: String): Boolean; function RegWriteMultiStringValue(const RootKey: Integer; const SubKeyName, ValueName, Data: String): Boolean; function RegWriteDWordValue(const RootKey: Integer; const SubKeyName, ValueName: String; const Data: Cardinal): Boolean; function RegWriteBinaryValue(const RootKey: Integer; const SubKeyName, ValueName: String; const Data: Cardinal): Boolean; function RegWriteBinaryValue(const RootKey: Integer; const SubKeyName, ValueName: String; const Data: AnsiString): Boolean;

function RegDeleteKeyIncludingSubkeys(const RootKey: Integer; const SubkeyName: String): Boolean; function RegDeleteKeyIfEmpty(const RootKey: Integer; const SubkeyName: String): Boolean; function <u>RegDeleteValue(const RootKey: Integer;</u> const SubKeyName, ValueName: String): Boolean;

INI File functions

```
function IniKeyExists(const Section, Key, Filename:
String): Boolean;
function IsIniSectionEmpty(const Section, Filename:
String): Boolean;
```

```
function GetIniBool(const Section, Key: String;
const Default: Boolean; const Filename: String):
Boolean
function GetIniInt(const Section, Key: String; const
Default, Min, Max: Longint; const Filename: String):
Longint;
function GetIniString(const Section, Key, Default,
Filename: String): String;
```

```
function <u>SetIniBool</u>(const Section, Key: String;
const Value: Boolean; const Filename: String):
Boolean;
function <u>SetIniInt</u>(const Section, Key: String; const
Value: Longint; const Filename: String): Boolean;
function <u>SetIniString</u>(const Section, Key, Value,
Filename: String): Boolean;
```

```
procedure DeleteIniSection(const Section, Filename:
String);
procedure DeleteIniEntry(const Section, Key,
Filename: String);
```

Custom Setup Wizard Page functions

function CreateInputQueryPage(const AfterID: Integer; const ACaption, ADescription, ASubCaption: String): TInputQueryWizardPage; function CreateInputOptionPage(const AfterID:

Integer; const ACaption, ADescription, ASubCaption: String; Exclusive, ListBox: Boolean): TInputOptionWizardPage; function CreateInputDirPage(const AfterID: Integer; const ACaption, ADescription, ASubCaption: String; AAppendDir: Boolean; ANewFolderName: String): TInputDirWizardPage; function <u>CreateInputFilePage(const AfterID: Integer;</u> const ACaption, ADescription, ASubCaption: String): TInputFileWizardPage; function <u>CreateOutputMsgPage(const AfterID: Integer;</u> const ACaption, ADescription, AMsg: String): TOutputMsgWizardPage; function <u>CreateOutputMsgMemoPage(const AfterID:</u> Integer; const ACaption, ADescription, ASubCaption: String; const AMsg: AnsiString): TOutputMsgMemoWizardPage;

function <u>CreateOutputProgressPage</u>(const ACaption, ADescription: String): TOutputProgressWizardPage; function <u>CreateCustomPage</u>(const AfterID: Integer; const ACaption, ADescription: String): TWizardPage;

function <u>CreateCustomForm</u>: TSetupForm;

function PageFromID(const ID: Integer): TWizardPage; function PageIndexFromID(const ID: Integer): Integer; function ScaleX(X: Integer): Integer; function ScaleY(Y: Integer): Integer;

Dialog functions

function MsgBox(const Text: String; const Typ: TMsgBoxType; const Buttons: Integer): Integer; function SuppressibleMsgBox(const Text: String; const Typ: TMsgBoxType; const Buttons, Default: Integer): Integer; function GetOpenFileName(const Prompt: String; var

```
FileName: String; const InitialDirectory, Filter,
DefaultExtension: String): Boolean;
function GetSaveFileName(const Prompt: String; var
FileName: String; const InitialDirectory, Filter,
DefaultExtension: String): Boolean;
function BrowseForFolder(const Prompt: String; var
Directory: String; const NewFolderButton: Boolean):
Boolean;
function ExitSetupMsqBox: Boolean;
```

COM Automation objects support functions

```
function CreateOleObject(const ClassName: string):
Variant;
function GetActiveOleObject(const ClassName:
string): Variant;
function IDispatchInvoke(Self: IDispatch;
PropertySet: Boolean; const Name: String; Par: array
of Variant): Variant;
function CreateComObject(const ClassID: TGUID):
IUnknown;
function StringToGUID(const S: String): TGUID;
procedure OleCheck(Result: HResult);
procedure CoFreeUnusedLibraries;
```

Setup Logging functions

procedure Log(const S: String);

Other functions

procedure <u>Sleep(const Milliseconds: LongInt);</u> function <u>Random(const Range: Integer): Integer;</u> procedure <u>Beep;</u>

procedure BringToFrontAndRestore;

Deprecated functions

function LoadDLL(const DLLName: String; var ErrorCode: Integer): Longint;

function <u>CallDLLProc</u>(const DLLHandle: Longint; const ProcName: String; const Param1, Param2: Longint; var Result: Longint): Boolean;

function FreeDLL(const DLLHandle: Longint): Boolean;

function <u>CastStringToInteger(var S: String):</u>
Longint;
function <u>CastIntegerToString(const L: Longint):</u>
String;

Constants

Here's the list of constants used by these functions:

CurStep values ssInstall, ssPostInstall, ssDone

CurPage values wpWelcome, wpLicense, wpPassword, wpInfoBefore, wpUserInfo, wpSelectDir, wpSelectComponents, wpSelectProgramGroup, wpSelectTasks, wpReady, wpPreparing, wpInstalling, wpInfoAfter, wpFinished

TMsgBoxType mbInformation, mbConfirmation, mbError, mbCriticalError

MsgBox - Buttons flags MB_OK, MB_OKCANCEL, MB_ABORTRETRYIGNORE, MB_YESNOCANCEL, MB_YESNO, MB_RETRYCANCEL, MB_DEFBUTTON1, MB_DEFBUTTON2, MB_DEFBUTTON3, MB_SETFOREGROUND

MsgBox - return values IDOK, IDCANCEL, IDABORT, IDRETRY, IDIGNORE, IDYES, IDNO

TShellFolderID

sfDesktop, sfStartMenu, sfPrograms, sfStartup, sfSendTo, sfFonts, sfAppData, sfDocs, sfTemplates, sfFavorites, sfLocalAppData

Reg* - RootKey values (also see the [Registry] section documentation) HKEY_CLASSES_ROOT, HKEY_CLASSES_ROOT_32, HKEY_CLASSES_ROOT_64, HKEY_CURRENT_USER, HKEY_CURRENT_USER_32, HKEY_CURRENT_USER_64, HKEY_LOCAL_MACHINE, HKEY_LOCAL_MACHINE_32, HKEY_LOCAL_MACHINE_64, HKEY_USERS, HKEY_USERS_32, HKEY_USERS_64, HKEY_PERFORMANCE_DATA, HKEY_CURRENT_CONFIG, HKEY_CURRENT_CONFIG_32, HKEY_CURRENT_CONFIG_64, HKEY_DYN_DATA, HKCR, HKCR32, HKCR64, HKCU, HKCU32, HKCU64, HKLM, HKLM32, HKLM64, HKU, HKU32, HKU64, HKCC, HKCC32, HKCC64

TShouldProcessEntryResult srNo, srYes, srUnknown

TSetupMessageID Use 'msg' + the message name. Example: SetupMessage(msgSetupAppTitle)

Exec and ShellExec - ShowCmd values SW_SHOW, SW_SHOWNORMAL, SW_SHOWMAXIMIZED, SW_SHOWMINIMIZED, SW_SHOWMINNOACTIVE, SW_HIDE

Pascal Scripting: Support Classes Reference

Below is the list of support classes that can be used from within the Pascal script. There are also three support objects available: MainForm of type TMainForm, WizardForm of type TWizardForm and UninstallProgressForm of type TUninstallProgressForm and one special constant: crHand of type TControl.Cursor. Note: MainForm is only visible if <u>WindowVisible</u> is set to yes.

Note: you may find it useful to also refer to the Delphi Visual Component Library (VCL) Help files by Embarcadero Technologies, since the classes below are mostly simple wrappers around the VCL classes Inno Setup uses internally. See <u>http://docs.embarcadero.com/products/rad_studio/</u>.

```
TObject = class
  constructor Create;
  procedure Free;
end;
TPersistent = class(TObject)
  procedure Assign(Source: <u>TPersistent</u>);
end;
TComponent = class(<u>TPersistent</u>)
  function FindComponent(AName: String): <u>TComponent;</u>
  constructor Create(AOwner: <u>TComponent</u>);
  property Owner: <u>TComponent;</u> read write;
  procedure DestroyComponents;
  procedure Destroying;
  procedure FreeNotification(AComponent:
TComponent);
  procedure InsertComponent(AComponent: <u>TComponent</u>);
  procedure RemoveComponent(AComponent: <u>TComponent</u>);
  property Components[Index: Integer]: <u>TComponent;</u>
read;
  property ComponentCount: Integer; read;
  property ComponentIndex: Integer; read write;
```

```
property ComponentState: Byte; read;
  property DesignInfo: Longint; read write;
  property Name: String; read write;
  property Tag: Longint; read write;
end;
TStrings = class(<u>TPersistent</u>)
  function Add(S: String): Integer;
  procedure Append(S: String);
  procedure AddStrings(Strings: <u>TStrings</u>);
  procedure Clear;
  procedure Delete(Index: Integer);
  function IndexOf(const S: String): Integer;
  procedure Insert(Index: Integer; S: String);
  property Count: Integer; read;
  property Text: String; read write;
  property CommaText: String; read write;
  procedure LoadFromFile(FileName: String);
  procedure SaveToFile(FileName: String);
  property Strings[Index: Integer]: String; read
write;
  property Objects[Index: Integer]: Tobject; read
write;
end;
TNotifyEvent = procedure(Sender: <u>TObject</u>);
TStringList = class(<u>TStrings</u>)
  function Find(S: String; var Index: Integer):
Boolean;
  procedure Sort;
  property Duplicates: TDuplicates; read write;
  property Sorted: Boolean; read write;
  property OnChange: <u>TNotifyEvent;</u> read write;
  property OnChanging: TNotifyEvent; read write;
end;
```
```
TStream = class(<u>TObject</u>)
  function Read(Buffer: String; Count: Longint):
Longint;
  function Write(Buffer: String; Count: Longint):
Longint;
  function Seek(Offset: Longint; Origin: Word):
Longint;
  procedure ReadBuffer(Buffer: String; Count:
Longint);
  procedure WriteBuffer(Buffer: String; Count:
Longint);
  function CopyFrom(Source: TStream; Count:
Longint): Longint;
  property Position: Longint; read write;
  property Size: Longint; read write;
end;
THandleStream = class(<u>TStream</u>)
  constructor Create(AHandle: Integer);
  property Handle: Integer; read;
end;
TFileStream = class(<u>THandleStream</u>)
  constructor Create(Filename: String; Mode: Word);
end;
TStringStream = class(<u>TStream</u>)
  constructor Create(AString: String);
end;
TGraphicsObject = class(<u>TPersistent</u>)
  property OnChange: TNotifyEvent; read write;
end;
TFontStyle = (fsBold, fsItalic, fsUnderline,
fsStrikeOut);
```

```
TFontStyles = set of <u>TFontStyle;</u>
```

```
TFont = class(<u>TGraphicsObject</u>)
  constructor Create;
  property Handle: Integer; read;
  property Color: Integer; read write;
  property Height: Integer; read write;
  property Name: String; read write;
  property Pitch: Byte; read write;
  property Size: Integer; read write;
  property PixelsPerInch: Integer; read write;
  property Style: <u>TFontStyles</u>; read write;
end;
TCanvas = class(<u>TPersistent</u>)
  procedure Arc(X1, Y1, X2, Y2, X3, Y3, X4, Y4:
Integer);
  procedure Chord(X1, Y1, X2, Y2, X3, Y3, X4, Y4:
Integer);
  procedure Draw(X, Y: Integer; Graphic: <u>TGraphic</u>);
  procedure Ellipse(X1, Y1, X2, Y2: Integer);
  procedure FloodFill(X, Y: Integer; Color: TColor;
FillStyle: Byte);
  procedure LineTo(X, Y: Integer);
  procedure MoveTo(X, Y: Integer);
  procedure Pie(X1, Y1, X2, Y2, X3, Y3, X4, Y4:
Integer);
  procedure Rectangle(X1, Y1, X2, Y2: Integer);
  procedure Refresh;
  procedure RoundRect(X1, Y1, X2, Y2, X3, Y3:
Integer);
  function TextHeight(Text: String): Integer;
  procedure TextOut(X, Y: Integer; Text: String);
  function TextWidth(Text: String): Integer;
  property Handle: Integer; read write;
  property Pixels: Integer Integer Integer; read
write;
```

```
property Brush: <u>TBrush;</u> read;
  property CopyMode: Byte; read write;
  property Font: TFont; read;
  property Pen: <u>TPen;</u> read;
end;
TPenMode = (pmBlack, pmWhite, pmNop, pmNot, pmCopy,
pmNotCopy, pmMergePenNot, pmMaskPenNot,
pmMergeNotPen, pmMaskNotPen, pmMerge, pmNotMerge,
pmMask, pmNotMask, pmXor, pmNotXor);
TPenStyle = (psSolid, psDash, psDot, psDashDot,
psDashDotDot, psClear, psInsideFrame);
TPen = class(<u>TGraphicsObject</u>)
  constructor Create;
  property Color: TColor; read write;
  property Mode: TPenMode; read write;
  property Style: TPenStyle; read write;
  property Width: Integer; read write;
end;
TBrushStyle = (bsSolid, bsClear, bsHorizontal,
bsVertical, bsFDiagonal, bsBDiagonal, bsCross,
bsDiagCross);
TBrush = class(TGraphicsObject)
  constructor Create;
  property Color: TColor; read write;
  property Style: TBrushStyle; read write;
end;
TGraphic = class(<u>TPersistent</u>)
  procedure LoadFromFile(const Filename: String);
  procedure SaveToFile(const Filename: String);
  property Empty: Boolean; read write;
  property Height: Integer; read write;
```

```
property Modified: Boolean; read write;
  property Width: Integer; read write;
  property OnChange: <u>TNotifyEvent;</u> read write;
end;
TBitmap = class(TGraphic)
  procedure LoadFromStream(Stream: <u>TStream</u>);
  procedure SaveToStream(Stream: <u>TStream</u>);
  property Canvas: <u>TCanvas</u>; read write;
  property Handle: HBITMAP; read write;
end;
TAlign = (alNone, alTop, alBottom, alLeft, alRight,
alClient);
TControl = class(TComponent)
  constructor Create(AOwner: <u>TComponent</u>);
  procedure BringToFront;
  procedure Hide;
  procedure Invalidate;
  procedure Refresh;
  procedure Repaint;
  procedure SendToBack;
  procedure Show;
  procedure Update;
  procedure SetBounds(ALeft, ATop, AWidth, AHeight:
Integer);
  property Left: Integer; read write;
  property Top: Integer; read write;
  property Width: Integer; read write;
  property Height: Integer; read write;
  property Hint: String; read write;
  property Align: <u>TAlign;</u> read write;
  property ClientHeight: Longint; read write;
  property ClientWidth: Longint; read write;
  property ShowHint: Boolean; read write;
  property Visible: Boolean; read write;
```

```
property Enabled: Boolean; read write;
  property Hint: String; read write;
  property Cursor: Integer; read write;
end;
TWinControl = class(TControl)
  property Parent: <u>TWinControl</u>; read write;
  property ParentBackground: Boolean; read write;
  property Handle: Longint; read write;
  property Showing: Boolean; read;
  property TabOrder: Integer; read write;
  property TabStop: Boolean; read write;
  function CanFocus: Boolean;
  function Focused: Boolean;
  property Controls[Index: Integer]: <u>TControl;</u> read;
  property ControlCount: Integer; read;
end;
TGraphicControl = class(TControl)
end;
TCustomControl = class(<u>TWinControl</u>)
end;
TScrollBarKind = (sbHorizontal, sbVertical);
TScrollBarInc = SmallInt;
TScrollingWinControl = class(<u>TWinControl</u>)
  procedure ScrollInView(AControl: <u>TControl</u>);
end;
TFormBorderStyle = (bsNone, bsSingle, bsSizeable,
bsDialog, bsToolWindow, bsSizeToolWin);
TBorderIcon = (biSystemMenu, biMinimize, biMaximize,
biHelp);
```

```
TBorderIcons = set of <u>TBorderIcon</u>;
TPosition = (poDesigned, poDefault,
poDefaultPosOnly, poDefaultSizeOnly, poScreenCenter,
poDesktopCenter, poMainFormCenter,
poOwnerFormCenter);
TCloseAction = (caNone, caHide, caFree, caMinimize);
TCloseEvent = procedure(Sender: <u>TObject;</u> var Action:
TCloseAction);
TCloseQueryEvent = procedure(Sender: <u>TObject;</u> var
CanClose: Boolean);
TEShiftState = (ssShift, ssAlt, ssCtrl, ssLeft,
ssRight, ssMiddle, ssDouble);
TShiftState = set of TEShiftState;
TKeyEvent = procedure(Sender: <u>TObject;</u> var Key:
Word; Shift: TShiftState);
TKeyPressEvent = procedure(Sender: <u>TObject;</u> var Key:
Char);
TForm = class(TScrollingWinControl)
  constructor CreateNew(AOwner: <u>TComponent</u>);
  procedure Close;
  procedure Hide;
  procedure Show;
  function ShowModal: Integer;
```

```
procedure Release;
```

```
property Active: Boolean; read;
property ActiveControl: <u>TWinControl</u>; read write;
```

```
property BorderIcons: TBorderIcons; read write;
```

```
property BorderStyle: <u>TFormBorderStyle;</u> read
write;
  property Caption: String; read write;
  property AutoScroll: Boolean; read write;
  property Color: TColor; read write;
  property Font: TFont; read write;
  property FormStyle: TFormStyle; read write;
  property KeyPreview: Boolean; read write;
  property Position: <a>TPosition</a>; read write;
  property OnActivate: <u>TNotifyEvent;</u> read write;
  property OnClick: <u>TNotifyEvent</u>; read write;
  property OnDblClick: TNotifyEvent; read write;
  property OnClose: TCloseEvent; read write;
  property OnCloseQuery: <u>TCloseQueryEvent;</u> read
write;
  property OnCreate: <u>TNotifyEvent;</u> read write;
  property OnDestroy: <u>TNotifyEvent;</u> read write;
  property OnDeactivate: <u>TNotifyEvent;</u> read write;
  property OnHide: <u>TNotifyEvent;</u> read write;
  property OnKeyDown: TKeyEvent; read write;
  property OnKeyPress: <u>TKeyPressEvent;</u> read write;
  property OnKeyUp: <u>TKeyEvent;</u> read write;
  property OnResize: TNotifyEvent; read write;
  property OnShow: <u>TNotifyEvent;</u> read write;
end;
TCustomLabel = class(TGraphicControl)
end;
TAlignment = (taLeftJustify, taRightJustify,
taCenter);
TLabel = class(TCustomLabel)
  property Alignment: <u>TAlignment</u>; read write;
  property AutoSize: Boolean; read write;
  property Caption: String; read write;
```

```
property Color: TColor; read write;
```

```
property FocusControl: <u>TWinControl</u>; read write;
property Font: <u>TFont</u>; read write;
property WordWrap: Boolean; read write;
property OnClick: <u>TNotifyEvent</u>; read write;
property OnDblClick: <u>TNotifyEvent</u>; read write;
end;
```

```
TCustomEdit = class(TWinControl)
  procedure Clear;
  procedure ClearSelection;
  procedure SelectAll;
  property Modified: Boolean; read write;
  property SelLength: Integer; read write;
  property SelStart: Integer; read write;
  property SelText: String; read write;
  property Text: String; read write;
end;
```

```
TBorderStyle = TFormBorderStyle;
```

```
TEditCharCase = (ecNormal, ecUpperCase,
ecLowerCase);
```

```
TEdit = class(<u>TCustomEdit</u>)
```

```
property AutoSelect: Boolean; read write;
property AutoSize: Boolean; read write;
property BorderStyle: <u>TBorderStyle</u>; read write;
property CharCase: <u>TEditCharCase</u>; read write;
property Color: TColor; read write;
property Font: <u>TFont</u>; read write;
property HideSelection: Boolean; read write;
property MaxLength: Integer; read write;
property PasswordChar: Char; read write;
property ReadOnly: Boolean; read write;
property Text: String; read write;
property OnChange: <u>TNotifyEvent</u>; read write;
```

```
property OnDblClick: <u>TNotifyEvent;</u> read write;
  property OnKeyDown: <u>TKeyEvent;</u> read write;
  property OnKeyPress: <u>TKeyPressEvent;</u> read write;
  property OnKeyUp: <u>TKeyEvent;</u> read write;
end;
TNewEdit = class(<u>TEdit</u>)
end;
TCustomMemo = class(<u>TCustomEdit</u>)
  property Lines: TStrings; read write;
end;
TScrollStyle = (ssNone, ssHorizontal, ssVertical,
ssBoth);
TMemo = class(<u>TCustomMemo</u>)
  property Lines: TStrings; read write;
  property Alignment: TAlignment; read write;
  property BorderStyle: TBorderStyle; read write;
  property Color: TColor; read write;
  property Font: TFont; read write;
  property HideSelection: Boolean; read write;
  property MaxLength: Integer; read write;
  property ReadOnly: Boolean; read write;
  property ScrollBars: TScrollStyle; read write;
  property WantReturns: Boolean; read write;
  property WantTabs: Boolean; read write;
  property WordWrap: Boolean; read write;
  property OnChange: <u>TNotifyEvent;</u> read write;
  property OnClick: TNotifyEvent; read write;
  property OnDblClick: TNotifyEvent; read write;
  property OnKeyDown: <u>TKeyEvent;</u> read write;
  property OnKeyPress: <u>TKeyPressEvent;</u> read write;
  property OnKeyUp: <u>TKeyEvent;</u> read write;
end;
```

```
TNewMemo = class(<u>TMemo</u>)
end;
TCustomComboBox = class(<u>TWinControl</u>)
  property DroppedDown: Boolean; read write;
  property Items: TStrings; read write;
  property ItemIndex: Integer; read write;
end;
TComboBoxStyle = (csDropDown, csSimple,
csDropDownList, csOwnerDrawFixed,
csOwnerDrawVariable);
TComboBox = class(<u>TCustomComboBox</u>)
  property Style: <u>TComboBoxStyle;</u> read write;
  property Color: TColor; read write;
  property DropDownCount: Integer; read write;
  property Font: TFont; read write;
  property MaxLength: Integer; read write;
  property Sorted: Boolean; read write;
  property Text: String; read write;
  property OnChange: <u>TNotifyEvent;</u> read write;
  property OnClick: TNotifyEvent; read write;
  property OnDblClick: TNotifyEvent; read write;
  property OnDropDown: <u>TNotifyEvent;</u> read write;
  property OnKeyDown: <u>TKeyEvent;</u> read write;
  property OnKeyPress: <u>TKeyPressEvent;</u> read write;
  property OnKeyUp: <u>TKeyEvent;</u> read write;
end;
TNewComboBox = class(<u>TComboBox</u>)
end;
TButtonControl = class(<u>TWinControl</u>)
end;
```

```
TButton = class(<u>TButtonControl</u>)
```

```
property Cancel: Boolean; read write;
  property Caption: String; read write;
  property Default: Boolean; read write;
  property Font: TFont; read write;
  property ModalResult: Longint; read write;
  property OnClick: <u>TNotifyEvent;</u> read write;
end;
TNewButton = class(<u>TButton</u>)
end;
TCustomCheckBox = class(TButtonControl)
end;
TCheckBoxState = (cbUnchecked, cbChecked, cbGrayed);
TCheckBox = class(<u>TCustomCheckBox</u>)
  property Alignment: <u>TAlignment</u>; read write;
  property AllowGrayed: Boolean; read write;
  property Caption: String; read write;
  property Checked: Boolean; read write;
  property Color: TColor; read write;
  property Font: TFont; read write;
  property State: <u>TCheckBoxState;</u> read write;
  property OnClick: TNotifyEvent; read write;
end;
TNewCheckBox = class(<u>TCheckBox</u>)
end;
TRadioButton = class(<u>TButtonControl</u>)
  property Alignment: <u>TAlignment</u>; read write;
  property Caption: String; read write;
  property Checked: Boolean; read write;
  property Color: TColor; read write;
  property Font: TFont; read write;
  property OnClick: <u>TNotifyEvent;</u> read write;
```

property OnDblClick: <u>TNotifyEvent;</u> read write; end;

```
TNewRadioButton = class(<u>TRadioButton</u>)
end;
TCustomListBox = class(<u>TWinControl</u>)
  property Items: TStrings; read write;
  property ItemIndex: Integer; read write;
  property SelCount: Integer; read;
  property Selected[Index: Integer]: Boolean; read
write;
end;
TListBoxStyle = (lbStandard, lbOwnerDrawFixed,
lbOwnerDrawVariable);
TListBox = class(<u>TCustomListBox</u>)
  property BorderStyle: TBorderStyle; read write;
  property Color: TColor; read write;
  property Font: TFont; read write;
  property MultiSelect: Boolean; read write;
  property Sorted: Boolean; read write;
  property Style: <u>TListBoxStyle;</u> read write;
  property OnClick: TNotifyEvent; read write;
  property OnDblClick: TNotifyEvent; read write;
  property OnKeyDown: <u>TKeyEvent;</u> read write;
  property OnKeyPress: <u>TKeyPressEvent;</u> read write;
  property OnKeyUp: <u>TKeyEvent;</u> read write;
end;
```

```
TNewListBox = class(<u>TListBox</u>)
end;
```

```
TBevelShape = (bsBox, bsFrame, bsTopLine,
bsBottomLine, bsLeftLine, bsRightLine, bsSpacer);
```

```
TBevelStyle = (bsLowered, bsRaised);
TBevel = class(TGraphicControl)
  property Shape: <u>TBevelShape;</u> read write;
  property Style: <u>TBevelStyle</u>; read write;
end;
TCustomPanel = class(TCustomControl)
end;
TPanelBevel = (bvNone, bvLowered, bvRaised,
bvSpace);
TBevelWidth = Longint;
TBorderWidth = Longint;
TPanel = class(TCustomPanel)
  property Alignment: TAlignment; read write;
  property BevelInner: TPanelBevel; read write;
  property BevelOuter: <u>TPanelBevel;</u> read write;
  property BevelWidth: <u>TBevelWidth</u>; read write;
  property BorderWidth: <a>TBorderWidth</a>; read write;
  property BorderStyle: <u>TBorderStyle;</u> read write;
  property Caption: String; read write;
  property Color: TColor; read write;
  property Font: TFont; read write;
  property OnClick: TNotifyEvent; read write;
  property OnDblClick: TNotifyEvent; read write;
end;
TNewStaticText = class(<u>TWinControl</u>)
  function AdjustHeight: Integer;
  property AutoSize: Boolean; read write;
  property Caption: String; read write;
  property Color: TColor; read write;
  property FocusControl: <u>TWinControl;</u> read write;
```

```
property Font: TFont; read write;
property ForceLTRReading: Boolean; read write;
property ShowAccelChar: Boolean; read write;
property WordWrap: Boolean; read write;
property OnClick: TNotifyEvent; read write;
property OnDblClick: TNotifyEvent; read write;
end;
```

```
TCheckItemOperation = (coUncheck, coCheck,
coCheckWithChildren);
```

```
TNewCheckListBox = class(<u>TCustomListBox</u>)
```

```
function AddCheckBox(const ACaption, ASubItem:
String; ALevel: Byte; AChecked, AEnabled,
AHasInternalChildren, ACheckWhenParentChecked:
Boolean; AObject: <u>TObject</u>): Integer;
```

```
function AddGroup(ACaption, ASubItem: String;
ALevel: Byte; AObject: <u>TObject</u>): Integer;
```

```
function AddRadioButton(const ACaption, ASubItem:
String; ALevel: Byte; AChecked, AEnabled: Boolean;
AObject: <u>TObject</u>): Integer;
```

function CheckItem(const Index: Integer; const
AOperation: <u>TCheckItemOperation</u>): Boolean;

```
property Checked[Index: Integer]: Boolean; read
write;
```

```
property State[Index: Integer]: <u>TCheckBoxState;</u>
read write;
```

property ItemCaption[Index: Integer]: String; read
write;

property ItemEnabled[Index: Integer]: Boolean; read write;

property ItemLevel[Index: Integer]: Byte; read;

property ItemObject[Index: Integer]: TObject; read
write;

property ItemSubItem[Index: Integer]: String; read
write;

property Flat: Boolean; read write;

```
property MinItemHeight: Integer; read write;
  property Offset: Integer; read write;
  property OnClickCheck: <u>TNotifyEvent;</u> read write;
  property BorderStyle: <u>TBorderStyle;</u> read write;
  property Color: TColor; read write;
  property Font: TFont; read write;
  property Sorted: Boolean; read write;
  property OnClick: TNotifyEvent; read write;
  property OnDblClick: TNotifyEvent; read write;
  property OnKeyDown: <u>TKeyEvent;</u> read write;
  property OnKeyPress: <u>TKeyPressEvent;</u> read write;
  property OnKeyUp: <u>TKeyEvent;</u> read write;
  property ShowLines: Boolean; read write;
  property WantTabs: Boolean; read write;
  property RequireRadioSelection: Boolean; read
write;
end;
TNewProgressBarState = (npbsNormal, npbsError,
npbsPaused);
TNewProgressBarStyle = (npbstNormal, npbstMarquee);
TNewProgressBar = class(<u>TWinControl</u>)
  property Min: Longint; read write;
  property Max: Longint; read write;
  property Position: Longint; read write;
  property State: TNewProgressBarState; read write;
  property Style: TNewProgressBarStyle; read write;
  property Visible: Boolean; read write;
end;
TRichEditViewer = class(<u>TMemo</u>)
  property RTFText: AnsiString; write;
  property UseRichEdit: Boolean; read write;
end;
```

```
TPasswordEdit = class(<u>TCustomEdit</u>)
  property AutoSelect: Boolean; read write;
  property AutoSize: Boolean; read write;
  property BorderStyle: <u>TBorderStyle;</u> read write;
  property Color: TColor; read write;
  property Font: TFont; read write;
  property HideSelection: Boolean; read write;
  property MaxLength: Integer; read write;
  property Password: Boolean; read write;
  property ReadOnly: Boolean; read write;
  property Text: String; read write;
  property OnChange: <u>TNotifyEvent;</u> read write;
  property OnClick: <u>TNotifyEvent;</u> read write;
  property OnDblClick: TNotifyEvent; read write;
  property OnKeyDown: <u>TKeyEvent;</u> read write;
  property OnKeyPress: <u>TKeyPressEvent;</u> read write;
  property OnKeyUp: <u>TKeyEvent;</u> read write;
end;
```

```
TCustomFolderTreeView = class(<u>TWinControl</u>)
    procedure ChangeDirectory(const Value: String;
    const CreateNewItems: Boolean);
```

procedure CreateNewDirectory(const ADefaultName: String);

```
property: Directory: String; read write;
end;
```

TFolderRenameEvent = procedure(Sender: <u>TCustomFolderTreeView</u>; var NewName: String; var Accept: Boolean);

```
TFolderTreeView = class(TCustomFolderTreeView)
    property OnChange: TNotifyEvent; read write;
    property OnRename: TFolderRenameEvent; read write;
end;
```

```
TStartMenuFolderTreeView =
```

```
class(<u>TCustomFolderTreeView</u>)
```

```
procedure SetPaths(const AUserPrograms,
ACommonPrograms, AUserStartup, ACommonStartup:
String);
property OnChange: <u>TNotifyEvent</u>; read write;
```

```
property OnRename: <u>TFolderRenameEvent</u>; read write;
end;
```

```
TBitmapImage = class(TGraphicControl)
  property AutoSize: Boolean; read write;
  property BackColor: TColor; read write;
  property Center: Boolean; read write;
  property Bitmap: TBitmap; read write;
  property ReplaceColor: TColor; read write;
  property ReplaceWithColor: TColor; read write;
  property Stretch: Boolean; read write;
  property OnClick: TNotifyEvent; read write;
  property OnDblClick: TNotifyEvent; read write;
end;
```

```
TNewNotebook = class(TWinControl)
function FindNextPage(CurPage: TNewNotebookPage;
GoForward: Boolean): TNewNotebookPage;
property PageCount: Integer; read write;
property Pages[Index: Integer]: TNewNotebookPage;
read;
property ActivePage: TNewNotebookPage; read write;
end;
```

```
TNewNotebookPage = class(<u>TCustomControl</u>)
  property Color: TColor; read write;
  property Notebook: <u>TNewNotebook</u>; read write;
  property PageIndex: Integer; read write;
end;
```

```
TWizardPageNotifyEvent = procedure(Sender:
TWizardPage);
```

```
TWizardPageButtonEvent = function(Sender:
TWizardPage): Boolean;
TWizardPageCancelEvent = procedure(Sender:
TWizardPage; var ACancel, AConfirm: Boolean);
TWizardPageShouldSkipEvent = function(Sender:
TWizardPage): Boolean;
TWizardPage = class(<u>TComponent</u>)
  property ID: Integer; read;
  property Caption: String; read write;
  property Description: String; read write;
  property Surface: <u>TNewNotebookPage</u>; read write;
  property SurfaceHeight: Integer; read write;
  property SurfaceWidth: Integer; read write;
  property OnActivate: <u>TWizardPageNotifyEvent;</u> read
write;
  property OnBackButtonClick:
TWizardPageButtonEvent; read write;
  property OnCancelButtonClick:
TWizardPageCancelEvent; read write;
  property OnNextButtonClick:
<u>TWizardPageButtonEvent;</u> read write;
  property OnShouldSkipPage:
TWizardPageShouldSkipEvent; read write;
end;
TInputQueryWizardPage = class(<u>TWizardPage</u>)
  function Add(const APrompt: String; const
APassword: Boolean): Integer;
  property Edits[Index: Integer]: <u>TPasswordEdit;</u>
read;
  property PromptLabels[Index: Integer]:
TNewStaticText; read;
  property SubCaptionLabel: <u>TNewStaticText;</u> read;
  property Values[Index: Integer]: String; read
write;
end;
```

```
TInputOptionWizardPage = class(<u>TWizardPage</u>)
  function Add(const ACaption: String): Integer;
  function AddEx(const ACaption: String; const
ALevel: Byte; const AExclusive: Boolean): Integer;
  property CheckListBox: <u>TNewCheckListBox;</u> read;
  property SelectedValueIndex: Integer; read write;
  property SubCaptionLabel: <u>TNewStaticText;</u> read;
  property Values[Index: Integer]: Boolean; read
write;
end;
TInputDirWizardPage = class(<u>TWizardPage</u>)
  function Add(const APrompt: String): Integer;
  property Buttons[Index: Integer]: <u>TNewButton;</u>
read;
  property Edits[Index: Integer]: TEdit; read;
  property PromptLabels[Index: Integer]:
TNewStaticText; read;
  property SubCaptionLabel: TNewStaticText; read;
  property Values[Index: Integer]: String; read
write;
end;
TInputFileWizardPage = class(<u>TWizardPage</u>)
  function Add(const APrompt, AFilter,
ADefaultExtension: String): Integer;
  property Buttons[Index: Integer]: <u>TNewButton;</u>
read;
  property Edits[Index: Integer]: <u>TEdit;</u> read;
  property PromptLabels[Index: Integer]:
TNewStaticText; read;
  property SubCaptionLabel: <u>TNewStaticText;</u> read;
  property Values[Index: Integer]: String; read
write;
  property IsSaveButton[Index: Integer]: Boolean;
read write;
```

end;

```
TOutputMsgWizardPage = class(TWizardPage)
  property MsgLabel: <u>TNewStaticText;</u> read;
end;
TOutputMsgMemoWizardPage = class(<u>TWizardPage</u>)
  property RichEditViewer: <u>TRichEditViewer;</u> read;
  property SubCaptionLabel: <u>TNewStaticText;</u> read;
end;
TOutputProgressWizardPage = class(<u>TWizardPage</u>)
  procedure Hide;
  property Msg1Label: <u>TNewStaticText;</u> read;
  property Msg2Label: <u>TNewStaticText;</u> read;
  property ProgressBar: <u>TNewProgressBar</u>; read;
  procedure SetProgress(const Position, Max:
Longint);
  procedure SetText(const Msq1, Msq2: String);
  procedure Show;
end;
TUIStateForm = class(TForm)
end;
TSetupForm = class(<u>TUIStateForm</u>)
  procedure Center;
  procedure CenterInsideControl(const Ctl:
TWinControl; const InsideClientArea: Boolean);
  procedure FlipControlsIfNeeded;
  property ControlsFlipped: Boolean; read;
  property FlipControlsOnShow: Boolean; read write;
  property RightToLeft: Boolean; read;
end;
TMainForm = class(<u>TSetupForm</u>)
```

```
procedure ShowAboutBox;
```

end;

```
TWizardForm = class(<u>TSetupForm</u>)
  property CancelButton: <u>TNewButton;</u> read;
  property NextButton: <u>TNewButton;</u> read;
  property BackButton: <u>TNewButton;</u> read;
  property Notebook1: TNotebook; read;
  property Notebook2: TNotebook; read;
  property WelcomePage: <u>TNewNotebookPage;</u> read;
  property InnerPage: <u>TNewNotebookPage;</u> read;
  property FinishedPage: TNewNotebookPage; read;
  property LicensePage: TNewNotebookPage; read;
  property PasswordPage: <u>TNewNotebookPage;</u> read;
  property InfoBeforePage: <u>TNewNotebookPage;</u> read;
  property UserInfoPage: <u>TNewNotebookPage;</u> read;
  property SelectDirPage: TNewNotebookPage; read;
  property SelectComponentsPage: <u>TNewNotebookPage;</u>
read;
  property SelectProgramGroupPage: <u>TNewNotebookPage;</u>
read;
  property SelectTasksPage: <u>TNewNotebookPage;</u> read;
  property ReadyPage: <u>TNewNotebookPage;</u> read;
  property PreparingPage: <u>TNewNotebookPage;</u> read;
  property InstallingPage: <u>TNewNotebookPage;</u> read;
  property InfoAfterPage: <u>TNewNotebookPage;</u> read;
  property DiskSpaceLabel: <u>TNewStaticText;</u> read;
  property DirEdit: TEdit; read;
  property GroupEdit: <u>TNewEdit;</u> read;
  property NoIconsCheck: <u>TNewCheckBox;</u> read;
  property PasswordLabel: <u>TNewStaticText;</u> read;
  property PasswordEdit: <u>TPasswordEdit</u>; read;
  property PasswordEditLabel: <u>TNewStaticText;</u> read;
  property ReadyMemo: <u>TNewMemo;</u> read;
  property TypesCombo: <u>TNewComboBox;</u> read;
  property Bevel: TBevel; read;
  property WizardBitmapImage: TBitmapImage; read;
  property WelcomeLabel1: <u>TNewStaticText;</u> read;
```

property InfoBeforeMemo: <u>TRichEditViewer;</u> read; property InfoBeforeClickLabel: <u>TNewStaticText;</u> read; property MainPanel: TPanel; read; property Bevel1: TBevel; read; property PageNameLabel: <u>TNewStaticText;</u> read; property PageDescriptionLabel: <u>TNewStaticText;</u> read; property WizardSmallBitmapImage: TBitmapImage; read; property ReadyLabel: <u>TNewStaticText;</u> read; property FinishedLabel: <u>TNewStaticText;</u> read; property YesRadio: <u>TNewRadioButton;</u> read; property NoRadio: <u>TNewRadioButton;</u> read; property WizardBitmapImage2: TBitmapImage; read; property WelcomeLabel2: <u>TNewStaticText;</u> read; property LicenseLabel1: <u>TNewStaticText;</u> read; property LicenseMemo: <u>TRichEditViewer</u>; read; property InfoAfterMemo: TRichEditViewer; read; property InfoAfterClickLabel: <u>TNewStaticText;</u> read; property ComponentsList: <u>TNewCheckListBox;</u> read; property ComponentsDiskSpaceLabel: <u>TNewStaticText;</u> read; property BeveledLabel: <u>TNewStaticText;</u> read; property StatusLabel: <u>TNewStaticText;</u> read; property FilenameLabel: <u>TNewStaticText;</u> read; property ProgressGauge: <u>TNewProgressBar</u>; read; property SelectDirLabel: <u>TNewStaticText;</u> read; property SelectStartMenuFolderLabel: TNewStaticText; read; property SelectComponentsLabel: <u>TNewStaticText;</u> read; property SelectTasksLabel: <u>TNewStaticText;</u> read; property LicenseAcceptedRadio: <u>TNewRadioButton;</u> read; property LicenseNotAcceptedRadio: <u>TNewRadioButton;</u> read;

property UserInfoNameLabel: <u>TNewStaticText;</u> read; property UserInfoNameEdit: <u>TNewEdit;</u> read; property UserInfoOrgLabel: <u>TNewStaticText;</u> read; property UserInfoOrgEdit: <u>TNewEdit;</u> read; property PreparingErrorBitmapImage: <u>TBitmapImage;</u> read; property PreparingLabel: <u>TNewStaticText;</u> read; property FinishedHeadingLabel: <u>TNewStaticText;</u> read; property UserInfoSerialLabel: <u>TNewStaticText;</u> read; property UserInfoSerialEdit: <u>TNewEdit;</u> read; property TasksList: <u>TNewCheckListBox;</u> read; property RunList: <u>TNewCheckListBox;</u> read; property DirBrowseButton: <u>TNewButton;</u> read; property GroupBrowseButton: <u>TNewButton;</u> read; property SelectDirBitmapImage: TBitmapImage; read; property SelectGroupBitmapImage: TBitmapImage; read; property SelectDirBrowseLabel: <u>TNewStaticText;</u> read; property SelectStartMenuFolderBrowseLabel: TNewStaticText; read; property PreparingYesRadio: <u>TNewRadioButton;</u> read; property PreparingNoRadio: <u>TNewRadioButton;</u> read; property PreparingMemo: <u>TNewMemo;</u> read; property CurPageID: Integer; read; function AdjustLabelHeight(ALabel: TNewStaticText): Integer; procedure IncTopDecHeight(AControl: <u>TControl;</u> Amount: Integer); property PrevAppDir: String; read; end; TUninstallProgressForm = class(<u>TSetupForm</u>) property OuterNotebook: <u>TNewNotebook;</u> read;

```
property InnerPage: <u>TNewNotebookPage;</u> read;
  property InnerNotebook: <u>TNewNotebook;</u> read;
  property InstallingPage: <u>TNewNotebookPage;</u> read;
  property MainPanel: TPanel; read;
  property PageNameLabel: <u>TNewStaticText</u>; read;
  property PageDescriptionLabel: <u>TNewStaticText;</u>
read;
  property WizardSmallBitmapImage: TBitmapImage;
read;
  property Bevel1: TBevel; read;
  property StatusLabel: TNewStaticText; read;
  property ProgressBar: <u>TNewProgressBar</u>; read;
  property BeveledLabel: <u>TNewStaticText;</u> read;
  property Bevel: TBevel; read;
  property CancelButton: <u>TNewButton;</u> read;
end;
See also:
```

function <u>CreateInputQueryPage(const AfterID:</u>
Integer; const ACaption, ADescription, ASubCaption:
String): TInputQueryWizardPage;
function <u>CreateInputOptionPage(const AfterID:</u>
Integer; const ACaption, ADescription, ASubCaption:
String; Exclusive, ListBox: Boolean):

TInputOptionWizardPage;

function <u>CreateInputDirPage(const AfterID: Integer;</u> const ACaption, ADescription, ASubCaption: String; AAppendDir: Boolean; ANewFolderName: String): TInputDirWizardPage;

function <u>CreateInputFilePage(const AfterID: Integer;</u> const ACaption, ADescription, ASubCaption: String): TInputFileWizardPage;

function <u>CreateOutputMsgPage</u>(const AfterID: Integer; const ACaption, ADescription, AMsg: String): TOutputMsgWizardPage;

function <u>CreateOutputMsgMemoPage(const AfterID:</u>
Integer; const ACaption, ADescription, ASubCaption:
String; const AMsg: AnsiString):

TOutputMsgMemoWizardPage;

function <u>CreateOutputProgressPage(const ACaption,</u> ADescription: String): TOutputProgressWizardPage; function <u>CreateCustomPage(const AfterID: Integer;</u> const ACaption, ADescription: String): TWizardPage; function <u>CreateCustomForm</u>: TSetupForm;

function PageFromID(const ID: Integer): TWizardPage; function <u>MinimizePathName(const Filename: String;</u> const Font: TFont; MaxLen: Integer): String;

Pascal Scripting: Using Custom Wizard Pages

The Pascal script allows you to add custom pages to Setup's wizard. This includes "pre-built" wizard pages for common queries and completely custom wizard pages with the controls of your choice.

To use custom wizard pages, first create them inside your InitializeWizard event function. You can either use pre-built pages created by the CreateInput...Page and CreateOutput...Page functions or "empty" pages created by the CreateCustomPage function. See <u>Support Functions</u> topic for a listing and explanation of all Create...Page functions.

After creating each page, you add controls to it, either by calling the special methods of the pre-built pages, or by manually creating controls on the page yourself.

Most of the Create...Page functions take a "page ID" as their first parameter; this identifies the existing page after which the newly created page should be placed. There are several ways to find the "page ID" of an existing page. The pages you create yourself have ID properties which hold their page IDs. Built-in wizard pages have predefined IDs. For example, for the *Welcome* wizard page this is wpWelcome. See the <u>Support Functions</u> topic for a listing of all predefined IDs.

After the custom wizard pages are created, Setup will show and handle them just as if they were built-in wizard pages. This includes the calling of all page related event functions such as NextButtonClick and ShouldSkipPage.

At any time during Setup you can retrieve the values entered by the user either by using the special properties of the pre-built pages, or by using the properties of the controls you created yourself.

Open the "CodeDlg.iss" script in the "Examples" subdirectory of your Inno Setup directory for an example of how to use pre-built custom wizard pages and event functions. Open the "CodeClasses.iss" script for an example of how to use completely custom wizard pages and controls.

Pascal Scripting: Using DLLs

The Pascal script can call functions inside external DLLs. This includes both standard Win32 API functions inside standard Windows DLLs and custom functions in custom made DLLs (how to make such a custom DLL is beyond the scope of this help file).

To be able to call a DLL function you should first write the function prototype as normal but instead of then writing the function body, you use the 'external' keyword to specify a DLL. If your function has for example prototype function A(B: Integer): Integer;, the following three forms are supported:

```
function A(B: Integer): Integer;
external '<dllfunctionname>@<dllfilename>';
function A(B: Integer): Integer;
external '<dllfunctionname>@<dllfilename> <callin
function A(B: Integer): Integer;
external '<dllfunctionname>@<dllfilename> <callin
```

The first form specifies that the DLL function should be called using default calling convention, which is 'stdcall'. All standard Win32 API functions use 'stdcall' just like most custom DLL functions.

The second form specifies that the DLL function should be called using a special calling convention. Valid calling conventions are: 'stdcall' (the default), 'cdecl', 'pascal' and 'register'.

The third form specifies additional one or more options for loading the DLL, separated by spaces:

delayload

Specifies that the DLL should be delay loaded. Normally the Pascal script checks at startup whether the DLL function can be called and if not, refuses to run. This does not happen if you specify delay loading using 'delayload'. Use delay loading if you want to call a DLL function for which you don't know whether it will actually be available at runtime: if the DLL function can't be called, the Pascal script will still run but throw an expection when you try to call the DLL function which you can catch to handle the absence of the DLL function.

loadwithalteredsearchpath

Specifies that the DLL should be loaded using the Windows flag LOAD_WITH_ALTERED_SEARCH_PATH, which, in essence, causes the loader to search for any dependent DLLs in the directory containing the DLL.

setuponly

Specifies that the DLL should only be loaded when the script is running from Setup.

uninstallonly

Specifies that the DLL should only be loaded when the script is running from Uninstall.

An example (of the second form) if the DLL function has name 'A2' inside the DLL, the DLL has name 'MyDII.dll' and the DLL function uses the 'stdcall' calling convention:

```
[Code]
function A(B: Integer): Integer;
external 'A2@MyDll.dll stdcall';
```

Constants may be used in the DLL filename.

During Setup, a special 'files:' prefix may also be used to instruct Setup to automatically extract one or more DLLs from the [Files] section before loading the first DLL. For example:

```
[Files]
Source: "MyDll.dll"; Flags: dontcopy
Source: "A.dll"; Flags: dontcopy
Source: "B.dll"; Flags: dontcopy
[Code]
procedure MyDllFunc(hWnd: Integer; lpText, lpCapt
```

external 'MyDllFunc@files:MyDll.dll stdcall';
procedure ADllFunc(hWnd: Integer; lpText, lpCapti
external 'ADllFunc@files:A.dll,B.dll stdcall load

If you use a 'files:' prefix and <u>solid compression</u> is enabled, be sure to list your DLLs at (or near) the top of the [Files] section. In order to extract an arbitrary file in a solid-compressed installation, Setup must first decompress all prior files (to a temporary buffer in memory). This can result in a substantial delay if a number of other files are listed above the specified file in the [Files] section.

Open the "CodeDII.iss" file in the "Examples" subdirectory in your Inno Setup directory for an example script using DLLs.

The "Examples" subdirectory also contains two custom DLL example projects, one for Microsoft Visual C++ and one for Borland Delphi.

Pascal Scripting: Using COM Automation objects

The Pascal script can access COM (also known as OLE or ActiveX) methods and properties via the COM Automation objects support. This allows you to access for example standard Windows COM servers, custom COM servers, Visual Basic ActiveX DLLs and .NET assemblies via COM Interop.

IDispatch based COM

There are two support functions to initialize IDispatch based COM Automation objects: <u>CreateOleObject</u> and <u>GetActiveOleObject</u>.

Use <u>CreateOleObject</u> to create a new COM object with the specified class name. This function returns a variable of type Variant if successful and throws an exception otherwise.

Use <u>GetActiveOleObject</u> to connect to an existing COM object with the specified class name. This function returns a variable of type Variant if successful and throws an exception otherwise. In case of some programs, this can be used to detect whether the program is running or not.

The value returned by <u>CreateOleObject</u> or <u>GetActiveOleObject</u> can then be used to access the properties and methods of the COM object. The access is done via 'late binding' which means it is not checked whether the methods or properties you're trying to access actually exist until Setup actually needs to at run time.

To access a property or method whose name is a reserved word, use <u>IDispatchInvoke</u>.

Open the "CodeAutomation.iss" file in the "Examples" subdirectory in your Inno Setup directory for an example script using IDispatch based COM Automation objects.

IUnknown based COM

If the IDispatch interface isn't implemented by the object, you can use the IUnknown based COM support.

To initialize IUnknown based COM Automation objects use <u>CreateComObject</u>.

The value returned by <u>CreateComObject</u> can then be used to access the methods of the COM object after casting it to the desired interface. The access is done via 'early binding' which means the desired interface needs to be defined in the script, unlike for IDispatch based COM support.

<u>StringToGUID</u> can be used to convert the string representation of a GUID into a 'real' GUID. Use <u>OleCheck</u> to check the return values of any method you call.

If you copy the interface definition from any existing Delphi source code, remove the brackets around the interface GUID string. Also remove any calling conventions, Inno Setup assumes 'stdcall'. If the interface contains any functions you won't call, you can replace these by dummies to avoid having to define any special types used by them.

Open the "CodeAutomation2.iss" file in the "Examples" subdirectory in your Inno Setup directory for an example script using IUnknown based COM Automation objects.

Note: IUnknown based COM support requires Unicode Inno Setup.

General

COM objects are released automatically when they go out of scope. There are no functions to 'destroy' or 'free' them.

If you are extracting a COM Automation library to a temporary location and want to be able to delete it after using it, make sure you no longer have any references to the library and then call <u>CoFreeUnusedLibraries</u>. This Windows function will then attempt to unload the library so you can delete it.

Unicode Inno Setup

Beginning with Inno Setup 5.3.0, there are two versions of Inno Setup available: Non Unicode Inno Setup and Unicode Inno Setup.

Key features of Unicode Inno Setup are its ability to display any language on any system regardless of the system code page, and its ability to work with Unicode filenames. One could consider Unicode Inno Setup as the new standard Inno Setup and Non Unicode Inno Setup as an old special Inno Setup for those who want the very smallest size possible.

If you don't remember which version you installed, click the "Inno Setup Compiler" shortcut created in the Start Menu. If the version number displayed in its title bar says "(a)" you are running Non Unicode Inno Setup. If it says "(u)" you are running Unicode Inno Setup.

For the most part the two versions are used identically, and any differences between them are noted throughout the help file. However, the following overview lists the primary differences:

- Unicode Inno Setup uses the existing ANSI .isl language files and you should not and may not convert these to Unicode or anything similar since it does so automatically during compilation using the LanguageCodePage setting of the language. However, you do need to convert existing [Messages] and [CustomMessages] entries in your .iss files to Unicode if the language used a special LanguageCodePage.
- The automatic conversion is also done for any language specific plain text ANSI LicenseFile, InfoBeforeFile, or InfoAfterFile used so you should not convert these either (but you may do so if you wish anyway, unlike ANSI .isl language files).
- The [Setup] directive ShowUndisplayableLanguages is ignored by Unicode Inno Setup.
- Unicode Inno Setup is compiled with Delphi 2009 instead of Delphi 2 and 3, leading to slightly larger files. The source code however is still compatible with Delphi 2 and 3, and a non Unicode version will

remain available.

- Existing installations of your programs done by non Unicode installers can be freely updated by Unicode installers, and vice versa.
- Unicode Pascal Scripting notes:
 - The Unicode compiler sees type 'String' as a Unicode string, and 'Char' as a Unicode character. Its 'AnsiString' type hasn't changed and still is an ANSI string. Its 'PChar' type has been renamed to 'PAnsiChar'.
 - The Unicode compiler is more strict about correct ';' usage: it no longer accepts certain missing ';' characters.
 - The new RemObjects PascalScript version used by the Unicode compiler supports Unicode, but not for its input source. This means it does use Unicode string types as said, but any literal Unicode characters in the script will be converted to ANSI. This doesn't mean you can't display Unicode strings: you can for example instead use encoded Unicode characters to build Unicode strings (like S := #\$0100 + #\$0101 + 'Aa';), or load the string from a file using LoadStringsFromFile, or use a {cm:...} constant.
 - Some support functions had their prototype changed: some parameters of CreateOutputMsgMemoPage, RegQueryBinaryValue, RegWriteBinaryValue, OemToCharBuff, CharToOemBuff, LoadStringFromfile, SaveStringToFile, and GetMD50fString are of type AnsiString now instead of String.
 - Added new SaveStringsToUTF8File, and GetMD50fUnicodeString support functions.
 - Added new 'Int64' type, supported by IntToStr. Also added new StrToInt64, StrToInt64Def, and GetSpaceOnDisk64 support functions.

- Added new TStringStream class.
- If you want to compile an existing script that imports ANSI Windows API calls with the Unicode compiler, either upgrade to the 'W' Unicode API call or change the parameters from 'String' or 'PChar' to 'AnsiString'. The 'AnsiString' approach will make your [Code] compatible with both the Unicode and the non Unicode version.
- Unicode Inno Setup supports UTF-8 encoded .iss files (but not UTF-16).
- Unicode Inno Setup supports UTF-8 and UTF-16LE encoded .txt files for LicenseFile, InfoBeforeFile, and InfoAfterFile.

Note: Unicode Inno Setup can only create Unicode installers and like wise the non Unicode version can only create non Unicode installers. If you want to be able to create both Unicode and non Unicode installers on one computer, you have to install both versions of Inno Setup into different folders.

Example Scripts

The Inno Setup Example Scripts are 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.
Frequently Asked Questions

The Frequently Asked Questions is now located in a separate document. Please click the "Inno Setup FAQ" shortcut created in the Start Menu when you installed Inno Setup, or open the "isfaq.htm" file in your Inno Setup directory.

For the most recent Frequently Asked Questions, go to <u>http://www.jrsoftware.org/isfaq.php</u>

Wizard Pages

Below is a list of all the wizard pages Setup may potentially display, and the conditions under which they are displayed.

• Welcome Shown by default, but can be disabled via <u>DisableWelcomePage</u>.

License Agreement

Shown if <u>LicenseFile</u> is set. Users may proceed to the next page only if the option "I accept the agreement" is selected.

Password

Shown if <u>Password</u> is set. Users may proceed to the next page only after entering the correct password.

Information Shown if InfoBeforeFile is set.

- User Information Shown if <u>UserInfoPage</u> is set to yes.
- Select Destination Location Shown by default, but can be disabled via <u>DisableDirPage</u>.

• Select Components Shown if there are any [Components] entries.

• Select Start Menu Folder

Shown if there are any [Icons] entries, but can be disabled via <u>DisableProgramGroupPage</u>.

• Select Tasks

Shown if there are any [Tasks] entries, unless the [Tasks] entries are all tied to components that were not selected on the *Select Components* page.

Ready to Install

Shown by default, but can be disabled via <u>DisableReadyPage</u>.

• Preparing to Install

Normally, Setup will never stop or pause on this page. The only time it will is if Setup determines it can't continue or if it detects

applications using files that need to be updated.

The former can happen if the PrepareToInstall event function returned an error or if one or more files specified in the [Files] and [InstallDelete] sections were queued (by some other installation) to be replaced or deleted on the next restart. In this case, it tells the user they need to restart their computer and then run Setup again. Note that this check is performed on silent installations too, but any messages are displayed in a message box instead of inside a wizard page.

The latter can happen if <u>CloseApplications</u> is set to yes.

- **Installing** Shown during the actual installation process.
- Information Shown if <u>InfoAfterFile</u> is set.
- Setup Completed Shown by default, but can be disabled in some cases via <u>DisableFinishedPage</u>.

Installation Order

Once the actual installation process begins, this is the order in which the various installation tasks are performed:

- [InstallDelete] is processed.
- The entries in [UninstallDelete] are stored in the uninstall log (which, at this stage, is stored in memory).
- The application directory is created, if necessary.
- [Dirs] is processed.
- A filename for the uninstall log is reserved, if necessary.
- [Files] is processed. (File registration does not happen yet.)
- [lcons] is processed.
- [INI] is processed.
- [Registry] is processed.
- Files that needed to be registered are now registered, unless the system needs to be restarted, in which case no files are registered until the system is restarted.
- The Add/Remove Programs entry for the program is created, if necessary.
- The entries in [UninstallRun] are stored in the uninstall log.
- The uninstaller EXE and log are finalized and saved to disk. After this is done, the user is forbidden from cancelling the install, and any subsequent errors will not cause what was installed before to be rolled back.
- [Run] is processed, except for entries with the postinstall flag, which get processed after the *Setup Completed* wizard page is shown.
- If <u>ChangesAssociations</u> was set to yes, file associations are refreshed now.

• If <u>ChangesEnvironment</u> was set to yes, other applications are notified at this point.

All entries are processed by the installer in the order they appear in a section.

Changes are undone by the uninstaller in the *opposite* order in which the installer made them. This is because the uninstall log is parsed from end to beginning.

In this example:

```
[INI]
Filename: "{win}\MYPROG.INI"; Section: "InstallSe
Filename: "{win}\MYPROG.INI"; Section: "InstallSe
```

the installer will first record the data for first entry's

uninsdeletesectionifempty flag in the uninstall log, create the key of the second entry, and then record the data for the uninsdeleteentry flag in the uninstall log. When the program is uninstalled, the uninstaller will first process the uninsdeleteentry flag, deleting the entry, and then the uninsdeletesectionifempty flag, deleting the section.

Note that the uninstaller processes [UninstallRun] and [UninstallDelete] entries in the same order they appear in the script (not in reverse order).

Install Mode: 32-bit vs. 64-bit

An installation can run in one of two modes: 32-bit or 64-bit.

64-bit mode is selected if the user is running a 64-bit version of Windows and the system's processor architecture is included in the value of the <u>ArchitecturesInstallIn64BitMode</u> [Setup] section directive. Otherwise, 32-bit mode is used.

How do the two modes of installation differ? Primarily, the differences lie in where things are installed by default.

In 32-bit mode:

- The System32 path returned by the {sys} constant maps to the 32-bit System directory by default.
- The {pf} constant is equivalent to {pf32}.
- The {cf} constant is equivalent to {cf32}.
- [Registry] writes to the 32-bit view by default.
- The {reg:...} constant reads the 32-bit view by default.
- The Reg* [Code] support functions access the 32-bit view by default.
- The useapppaths flag of the [Icons] section reads the "App Paths" key in the 32-bit view of the registry.
- The regserver and regtypelib flags of the [Files] section load and register files inside a 32-bit process by default.
- The sharedfile flag of the [Files] section updates the "SharedDLLs" key in the 32-bit view of the registry by default.
- The Uninstall key is created in the 32-bit view of the registry.

In 64-bit mode:

• The System32 path returned by the {sys} constant maps to the 64-bit System directory by default when used in the [Dirs], [Files], [InstallDelete], [Run], [UninstallDelete], and [UninstallRun]

sections. This is because Setup/Uninstall temporarily disables <u>WOW64 file system redirection</u> when files/directories are accessed by those sections. Elsewhere, System32 and {sys} map to the 32-bit System directory, as is normal in a 32-bit process.

- The {pf} constant is equivalent to {pf64}.
- The {cf} constant is equivalent to {cf64}.
- [Registry] writes to the 64-bit view by default.
- The {reg:...} constant reads the 64-bit view by default.
- The Reg* [Code] support functions access the 64-bit view by default.
- The useapppaths flag of the [Icons] section reads the "App Paths" key in the 64-bit view of the registry.
- The regserver and regtypelib flags of the [Files] section load and register files inside a 64-bit process by default.
- The sharedfile flag of the [Files] section updates the "SharedDLLs" key in the 64-bit view of the registry by default.
- The Uninstall key is created in the 64-bit view of the registry.

64-bit Installation Limitations

Because Inno Setup is a 32-bit application, there are some limitations to be aware of when utilizing its 64-bit installation features:

- The System32 path returned by the {sys} constant does not always map to the 64-bit System directory. When Setup/Uninstall is running in 64-bit mode, it maps to the 64-bit System directory when used in the [Dirs], [Files], [InstallDelete], [Run], [UninstallDelete], and [UninstallRun] sections because Setup temporarily disables <u>WOW64 file system redirection</u> when files/directories are accessed by those sections. Elsewhere, System32 and {sys} map to the 32-bit System directory, as is normal in a 32-bit process.
- In the [Code] section, when Setup/Uninstall is running in 64-bit mode, functions that access files disable WOW64 file system redirection (unless overridden by a call to <u>EnableFsRedirection</u>). However, there are exceptions, listed below. These functions never disable file system redirection, meaning you cannot pass them (or get back) the name of a file located in the 64-bit System directory:

Ini	(all of the functions that manipulate .INI files)
BrowseForFolder	
CreateShellLink	
GetOpenFileName	
LoadDLL	(see following point)
ModifyPifFile	
SetCurrentDir	
ShellExec	(use Exec instead)
UnregisterFont	

Additionally, no VCL classes are capable of disabling file system redirection. For example, you cannot call the LoadFromFile method of TBitmap to load a bitmap file from the 64-bit System directory.

• You cannot load/use 64-bit DLLs in the [Code] section, because

Windows does not allow 32-bit processes to load 64-bit DLLs (and vice versa). A 32-bit process can, however, launch 64-bit EXEs. Use the Exec function or the [Run] section to do that.

Miscellaneous Notes

• To easily auto update your application, first make your application somehow detect a new version of your Setup.exe and make it locate or download this new version. Then, to auto update, start your Setup.exe from your application with for example the following command line:

```
/SP- /silent /noicons "/dir=expand:{pf}\My Prc
```

After starting setup.exe, exit your application as soon as possible. Note that to avoid problems with updating your .exe, Setup has an auto retry feature.

Optionally you could also use the skipifsilent and skipifnotsilent flags and make your application aware of a '/updated' parameter to for example show a nice message box to inform the user that the update has completed.

Command Line Compiler Execution

• Scripts can also be compiled by the Setup Compiler from the command line. Command line usage is as follows:

compil32 /cc <script name>

Example:

compil32 /cc "c:\isetup\samples\my script.iss"

As shown in the example above, filenames that include spaces must be enclosed in quotes.

Running the Setup Compiler from the command line does not suppress the normal progress display or any error messages. The Setup Compiler will return an exit code of 0 if the compile was successful, 1 if the command line parameters were invalid, or 2 if the compile failed.

• Alternatively, you can compile scripts using the console-mode compiler, ISCC.exe. Command line usage is as follows:

iscc [options] <script name>

Or to read from standard input:

```
iscc [options] -
```

Example:

```
iscc "c:\isetup\samples\my script.iss"
```

As shown in the example above, filenames that include spaces must be enclosed in quotes.

Valid options are: "/O-" to disable output (overriding any Output setting in the script), "/O+" to enable output (overriding any Output setting in the script), "/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), "/S" to specify a Sign Tool (any Sign Tools configured using the IDE will be specified automatically), "/Q[p]" for quiet compile (print only error messages, "p" will show progress info), and "/?" to show a help screen.

Example:

iscc /Qp /O"My Output" /F"MyProgram-1.0" /Sbyparam=\$p
"c:\isetup\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.

• The Setup Script Wizard can be started from the command line. Command line usage is as follows:

compil32 /wizard <wizard name> <script name>

Example:

compil32 /wizard "MyProg Script Wizard" "c:\temp.iss"

As shown in the example above, wizard names and filenames that include spaces must be enclosed in quotes.

Running the wizard from the command line does not suppress any error messages. The Setup Script Wizard will return an exit code of 0 if there was no error and additionally it will save the generated script file to the specified filename, 1 if the command line parameters were invalid, or 2 if the generated script file could not be saved. If the user cancelled the Setup Script Wizard, an exit code of 0 is returned and no script file is saved.

Setup Command Line Parameters

The Setup program accepts optional command line parameters. These can be useful to system administrators, and to other programs calling the Setup program.

/HELP, /?

Shows a summary of this information. Ignored if the UseSetupLdr [Setup] section directive was set to no.

/SP-

Disables the *This will install...* Do you wish to continue? prompt at the beginning of Setup. Of course, this will have no effect if the DisableStartupPrompt [Setup] section directive was set to yes.

/SILENT, /VERYSILENT

Instructs Setup to be silent or very silent. When Setup is silent the wizard and the background window are not displayed but the installation progress window is. When a setup is very silent this installation progress window is not displayed. Everything else is normal so for example error messages during installation are displayed and the startup prompt is (if you haven't disabled it with DisableStartupPrompt or the '/SP-' command line option explained above).

If a restart is necessary and the '/NORESTART' command isn't used (see below) and Setup is silent, it will display a *Reboot now*? message box. If it's very silent it will reboot without asking.

/SUPPRESSMSGBOXES

Instructs Setup to suppress message boxes. Only has an effect when combined with '/SILENT' or '/VERYSILENT'.

The default response in situations where there's a choice is:

- Yes in a 'Keep newer file?' situation.
- No in a 'File exists, confirm overwrite.' situation.
- Abort in Abort/Retry situations.
- Cancel in Retry/Cancel situations.

- Yes (=continue) in a DiskSpaceWarning/DirExists/DirDoesntExist/NoUninstallWarnin situation.
- Yes (=restart) in a FinishedRestartMessage/UninstalledAndNeedsRestart situation.

5 message boxes are not suppressible:

- The About Setup message box.
- The Exit Setup? message box.
- The FileNotInDir2 message box displayed when Setup requires a new disk to be inserted and the disk was not found.
- Any (error) message box displayed before Setup (or Uninstall) could read the command line parameters.
- Any message box displayed by [Code] support function MsgBox.

/LOG

Causes Setup to create a log file in the user's TEMP directory detailing file installation and [Run] actions taken during the installation process. This can be a helpful debugging aid. For example, if you suspect a file isn't being replaced when you believe it should be (or vice versa), the log file will tell you if the file was really skipped, and why.

The log file is created with a unique name based on the current date. (It will not overwrite or append to existing files.)

The information contained in the log file is technical in nature and therefore not intended to be understandable by end users. Nor is it designed to be machine-parseable; the format of the file is subject to change without notice.

/LOG="filename"

Same as /LOG, except it allows you to specify a fixed path/filename to use for the log file. If a file with the specified name already exists it will be overwritten. If the file cannot be created, Setup will abort with an error message.

/NOCANCEL

Prevents the user from cancelling during the installation process, by disabling the Cancel button and ignoring clicks on the close button. Useful along with '/SILENT' or '/VERYSILENT'.

/NORESTART

Prevents Setup from restarting the system following a successful installation, or after a *Preparing to Install* failure that requests a restart. Typically used along with /SILENT or /VERYSILENT.

RESTARTEXITCODE=*exit* code

Specifies a custom exit code that Setup is to return when the system needs to be restarted following a successful installation. (By default, 0 is returned in this case.) Typically used along with /NORESTART. See also: <u>Setup Exit Codes</u>

/CLOSEAPPLICATIONS

Instructs Setup to close applications using files that need to be updated by Setup if possible.

/NOCLOSEAPPLICATIONS

Prevents Setup from closing applications using files that need to be updated by Setup. If /CLOSEAPPLICATIONS was also used, this command line parameter is ignored.

(RESTARTAPPLICATIONS

Instructs Setup to restart applications if possible. If Setup didn't close these applications (for example because /NOCLOSEAPPLICATIONS was used), this command line parameter is ignored.

/NORESTARTAPPLICATIONS

Prevents Setup from restarting applications. If /RESTARTAPPLICATIONS was also used, this command line parameter is ignored.

/LOADINF="filename"

Instructs Setup to load the settings from the specified file after having checked the command line. This file can be prepared using the '/SAVEINF=' command as explained below.

Don't forget to use quotes if the filename contains spaces.

/SAVEINF="filename"

Instructs Setup to save installation settings to the specified file.

Don't forget to use quotes if the filename contains spaces.

/LANG=language

Specifies the language to use. *language* specifies the internal name of the language as specified in a [Languages] section entry.

When a valid /LANG parameter is used, the *Select Language* dialog will be suppressed.

/DIR="x:\dirname"

Overrides the default directory name displayed on the *Select Destination Location* wizard page. A fully qualified pathname must be specified. May include an "expand:" prefix which instructs Setup to expand any constants in the name. For example: '/DIR=expand: {pf}\My Program'.

/GROUP="folder name"

Overrides the default folder name displayed on the *Select Start Menu Folder* wizard page. May include an "expand:" prefix, see '/DIR='. If the [Setup] section directive

DisableProgramGroupPage was set to yes, this command line parameter is ignored.

/NOICONS

Instructs Setup to initially check the *Don't create a Start Menu folder* check box on the *Select Start Menu Folder* wizard page.

/TYPE=type name

Overrides the default setup type.

If the specified type exists and isn't a custom type, then any /COMPONENTS parameter will be ignored.

/COMPONENTS="comma separated list of component names"

Overrides the default <u>component</u> settings. Using this command line parameter causes Setup to automatically select a custom type. If no custom type is defined, this parameter is ignored. Only the specified components will be selected; the rest will be deselected.

If a component name is prefixed with a "*" character, any child components will be selected as well (except for those that include the dontinheritcheck flag). If a component name is prefixed with a "!" character, the component will be deselected.

This parameter does not change the state of components that include the fixed flag.

Example:

Deselect all components, then select the "help" and "plugins" components:

/COMPONENTS="help,plugins"

Example:

Deselect all components, then select a parent component and all of its children with the exception of one: /COMPONENTS="*parent,!parent\child"

ITASKS="comma separated list of task names"

Specifies a list of <u>tasks</u> that should be initially selected.

Only the specified tasks will be selected; the rest will be deselected. Use the /MERGETASKS parameter instead if you want to keep the default set of tasks and only select/deselect some of them.

If a task name is prefixed with a "*" character, any child tasks will be selected as well (except for those that include the dontinheritcheck flag). If a task name is prefixed with a "!" character, the task will be deselected.

Example:

Deselect all tasks, then select the "desktopicon" and "fileassoc" tasks:

/TASKS="desktopicon,fileassoc"

Example:

Deselect all tasks, then select a parent task and all of its children with the exception of one: /TASKS="*parent,!parent\child"

/MERGETASKS="comma separated list of task names"

Like the /TASKS parameter, except the specified tasks will be merged with the set of tasks that would have otherwise been selected by default.

If <u>UsePreviousTasks</u> is set to yes, the specified tasks will be selected/deselected after any previous tasks are restored.

Example:

Keep the default set of selected tasks, but additionally select the "desktopicon" and "fileassoc" tasks:

/MERGETASKS="desktopicon,fileassoc"

Example:

Keep the default set of selected tasks, but deselect the "desktopicon" task: /MERGETASKS="!desktopicon"

/PASSWORD=password

Specifies the password to use. If the [Setup] section directive Password was not set, this command line parameter is ignored.

When an invalid password is specified, this command line parameter is also ignored.

Setup Exit Codes

Beginning with Inno Setup 3.0.3, the Setup program may return one of the following exit codes:

- **0** Setup was successfully run to completion or the /HELP or /? command line parameter was used.
- **1** Setup failed to initialize.
- 2 The user clicked Cancel in the wizard before the actual installation started, or chose "No" on the opening "This will install..." message box.
- **3** A fatal error occurred while preparing to move to the next installation phase (for example, from displaying the pre-installation wizard pages to the actual installation process). This should never happen except under the most unusual of circumstances, such as running out of memory or Windows resources.
- 4 A fatal error occurred during the actual installation process.

Note: Errors that cause an Abort-Retry-Ignore box to be displayed are not fatal errors. If the user chooses *Abort* at such a message box, exit code 5 will be returned.

- **5** The user clicked Cancel during the actual installation process, or chose *Abort* at an Abort-Retry-Ignore box.
- 6 The Setup process was forcefully terminated by the debugger (*Run* | *Terminate* was used in the IDE).
- 7 The *Preparing to Install* stage determined that Setup cannot proceed with installation. (*First introduced in Inno Setup 5.4.1.*)
- 8 The *Preparing to Install* stage determined that Setup cannot proceed with installation, and that the system needs to be restarted in order to correct the problem. (*First introduced in Inno Setup 5.4.1.*)

Before returning an exit code of 1, 3, 4, 7, or 8, an error message explaining the problem will normally be displayed.

Future versions of Inno Setup may return additional exit codes, so applications checking the exit code should be programmed to handle unexpected exit codes gracefully. Any non-zero exit code indicates that Setup was not run to completion.

Uninstaller Command Line Parameters

The uninstaller program (unins???.exe) accepts optional command line parameters. These can be useful to system administrators, and to other programs calling the uninstaller program.

/SILENT, /VERYSILENT

When specified, the uninstaller will not ask the user for startup confirmation or display a message stating that uninstall is complete. Shared files that are no longer in use are deleted automatically without prompting. Any critical error messages will still be shown on the screen. When '/VERYSILENT' is specified, the uninstallation progress window is not displayed.

If a restart is necessary and the '/NORESTART' command isn't used (see below) and '/VERYSILENT' is specified, the uninstaller will reboot without asking.

/SUPPRESSMSGBOXES

Instructs the uninstaller to suppress message boxes. Only has an effect when combined with '/SILENT' and '/VERYSILENT'. See '/SUPPRESSMSGBOXES' under <u>Setup Command Line</u> <u>Parameters</u> for more details.

/LOG

Causes Uninstall to create a log file in the user's TEMP directory detailing file uninstallation and [UninstallRun] actions taken during the uninstallation process. This can be a helpful debugging aid.

The log file is created with a unique name based on the current date. (It will not overwrite or append to existing files.)

The information contained in the log file is technical in nature and therefore not intended to be understandable by end users. Nor is it designed to be machine-parseable; the format of the file is subject to change without notice.

/LOG="filename"

Same as /LOG, except it allows you to specify a fixed path/filename to use for the log file. If a file with the specified name already exists it will be overwritten. If the file cannot be created,

Uninstall will abort with an error message.

/NORESTART

Instructs the uninstaller not to reboot even if it's necessary.

Uninstaller Exit Codes

Beginning with Inno Setup 4.0.8, the uninstaller will return a non-zero exit code if the user cancels or a fatal error is encountered. Programs checking the exit code to detect failure should not check for a specific non-zero value; any non-zero exit code indicates that the uninstaller was not run to completion.

Note that at the moment you get an exit code back from the uninstaller, some code related to uninstallation might still be running. Because Windows doesn't allow programs to delete their own EXEs, the uninstaller creates and spawns a copy of itself in the TEMP directory. This "clone" performs the actual uninstallation, and at the end, terminates the original uninstaller EXE (at which point you get an exit code back), deletes it, then displays the "uninstall complete" message box (if it hasn't been suppressed with /SILENT or /VERYSILENT).

Unsafe Files

As a convenience to new users who are unfamiliar with which files they should and should not distribute, the Inno Setup compiler will display an error message if one attempts to install certain "unsafe" files using the [Files] section. These files are listed below.

(Note: It is possible to disable the error message by using a certain flag on the [Files] section entry, but this is NOT recommended.)

Any DLL file from own Windows System directory

You should not deploy any DLLs out of your own Windows System directory to {sys} because most of them are tailored for your own specific version of Windows, and will not work when installed on other versions. Often times a user's system will be **rendered unbootable** if you install a DLL from a different version of Windows. Another reason why it's a bad idea is that when you install programs on your computer, the DLLs may be replaced with different/incompatible versions without your knowledge. This could lead to unexpected and difficult-to-trace problems on users' systems when you build new installations.

Instead of deploying the DLLs from your Windows System directory, you should find versions that are specifically deemed "redistributable". Redistributable DLLs typically work on more than one version of Windows. To find redistributable versions of the Visual Basic and Visual C++ run-time DLLs, see the Inno Setup FAQ.

If you have a DLL residing in the Windows System directory that you are **absolutely sure** is redistributable, copy it to your script's source directory and deploy it from there instead.

ADVAPI32.DLL, COMDLG32.DLL, GDI32.DLL, KERNEL32.DLL, RICHED32.DLL, SHELL32.DLL, USER32.DLL, UXTHEME.DLL

These are all core components of Windows and must never be deployed with an installation. Users may only get new versions of these DLLs by installing a new version of Windows or a service pack or hotfix for Windows.

(Special case) COMCAT.DLL, MSVBVM50.DLL, MSVBVM60.DLL, OLEAUT32.DLL, OLEPRO32.DLL, STDOLE2.TLB

If DestDir is set to a location *other* than {sys} and the regserver or regtypelib flag is used, then the above files will be considered "unsafe". These files must never be deployed to and registered in a directory other than {sys} because doing so can potentially cause *all* programs on the system to use them in favor of the files in {sys}. Problems would result if your copies of the files are older than the ones in {sys}. Also, if your copies of the files were removed, other applications would break.

COMCTL32.DLL

Microsoft does not allow separate redistribution of COMCTL32.DLL (and for good reason - the file differs between platforms), so you should never place COMCTL32.DLL in a script's [Files] section. You can however direct your users to <u>download the COMCTL32</u> <u>update from Microsoft</u>, or distribute the COMCTL32 update along with your program.

SHDOCVW.DLL, SHLWAPI.DLL, URLMON.DLL, WININET.DLL

These are core components of Internet Explorer and are also used by Windows Explorer. Replacing them may prevent Explorer from starting. If your application depends on these DLLs, or a recent version of them, then your users will need to install a recent version of Internet Explorer to get them.

MSCOREE.DLL

This file is part of the Microsoft .NET Framework. You cannot safely install or update the .NET Framework by including this file with your installation. Call or direct your users to dotnetfx.exe instead.

Credits

The following is a list of those who have contributed significant code to the Inno Setup project, or otherwise deserve special recognition:

- Jean-loup Gailly & Mark Adler: Creators of the <u>zlib</u> r compression library that Inno Setup uses.
- Julian Seward: Creator of the <u>bzlib</u> r compression library that Inno Setup uses.
- Igor Pavlov: Creator of the <u>LZMA</u> r compression library that Inno Setup uses.
- ?: Most of the disk spanning code (1.09). (Sorry, somehow your name was lost!)
- Vince Valenti: Most of the code for the "Window" [Setup] section directives (1.12.4).
- Joe White: Code for ChangesAssociations [Setup] section directive (1.2.?).
- Jason Olsen: Most of the code for <u>appending to existing uninstall</u> <u>logs</u> (1.3.0).
- Martijn Laan: Code for Rich Edit 2.0 & URL detection support (1.3.13); silent uninstallation (1.3.25); system image list support in drive and directory lists (1.3.25); silent installation (2.0.0); [Types], [Components] and [Tasks] sections (2.0.0); postinstall flag (2.0.0); [Code] section (4.0.0); Subcomponents and subtasks support (4.0.0); Many other features after 4.0.0.
- Alex Yackimoff: Portions of <u>TNewCheckListBox</u> (4.0.0).
- Carlo Kok: <u>RemObjects Pascal Script</u> (4.0.0).
- Creators of <u>SynEdit</u>
 ^{second}: The syntax-highlighting editor used in the Compiler (2.0.0).
- Creators of <u>UniSynEdit</u>: The syntax-highlighting editor used in the Compiler (5.3.0).

- Creators of <u>Scintilla</u> : The syntax-highlighting editor used in the Compiler (5.4.0).
- glyEX ■: The Inno Setup logo, the compiler icon, the document icon, the Inno Setup installer wizard images and the images for the IDE's toolbar. 128x128 and 256x256 icon sizes by Motaz Alnuweiri.
- Zaher Dirkey: Initial work on improved right-to-left languages support.
- Evgeny Karpov of RemObjects Software: Initial work on Unicode support.

Support Inno Setup

To support Inno Setup, go to this page: http://www.jrsoftware.org/isdonate.php

Windows Versions

5.0.2195	Windows 2000
5.1.2600	Windows XP
	or Windows XP 64-Bit Edition Version 2002 (Itanium)
5.2.3790	Windows Server 2003
	or Windows XP x64 Edition (AMD64/EM64T)
	or Windows XP 64-Bit Edition Version 2003 (Itanium)
6.0.6000	Windows Vista
6.0.6001	Windows Vista with Service Pack 1
	or Windows Server 2008
6.1.7600	Windows 7
	or Windows Server 2008 R2
6.1.7601	Windows 7 with Service Pack 1
	or Windows Server 2008 R2 with Service Pack 1
6.2.9200	Windows 8
	or Windows Server 2012
6.3.9200	Windows 8.1
	or Windows Server 2012 R2
6.3.9600	Windows 8.1 with Update 1
10.0.10240	Windows 10

Note that there is normally no need to specify the build numbers (i.e., you may simply use "6.2" for Windows 8).

Using Build Number and/or Service Pack Levels

The versions specified in MinVersion and OnlyBelowVersion can optionally include build numbers and/or service pack levels.

Examples: 5.0.2195 5.0sp4 5.0.2195sp4

If a build number is not specified or is zero, Setup will not check the system's build number.

If a service pack level is not specified or is zero, Setup will not check the system's service pack level.

When a service pack level is specified, Setup will only compare it against the system's service pack level if the specified major and minor versions match the system's version. For example, if MinVersion specifies 5.0sp4, Setup will only check for SP4 on Windows 2000 (5.0) systems.

In an OnlyBelowVersion parameter, if the specified version matches the system's version, then Setup will normally consider the system's version to be too high. However, when a service pack level is specified, the specified version is allowed to match the system's version. For example, on Windows 2000 SP4, values of 5.0 and 5.0.2195 will fail the OnlyBelowVersion test, but 5.0sp5 and 5.0.2195sp5 will pass (as SP4 < sp5).

Notes on "yes" and "no"

For compatibility with previous Inno Setup versions, 1 and 0 may be used in place of yes and no, respectively.

Additionally, it allows true and false to be used in place of yes and no.

Pascal Scripting: WizardNolcons

Prototype:

function WizardNoIcons: Boolean;

Description:

Returns the current setting of the *Don't create a Start Menu folder* check box on the *Select Start Menu Folder* page of the wizard.

Appending to Existing Uninstall Logs

When a new version of an application is installed over an existing version, instead of creating a new uninstall log file (unins???.dat), Setup will by default look for and append to an existing uninstall log file that belongs to the <u>same application</u> and is in the same directory. This way, when the application is uninstalled, changes made by all the different installations will be undone (starting with the most recent installation).

The uninstaller will use the <u>messages</u> from the most recent installation of the application. However, there is an exception: if an installation was built with an older version of Inno Setup that included an older version of the uninstaller than the existing one on the user's system, neither the existing uninstaller nor its messages will be replaced. In this case the uninstall log will still be appended to, though, since the file format is backward compatible.

The application name displayed in the uninstaller will be the same as the value of the [Setup] section directive <u>AppName</u> from the most recent installation, unless <u>UpdateUninstallLogAppName</u> is set to no.

The uninstall log-appending feature is new to Inno Setup 1.3. If you wish to disable it, set the [Setup] section directive <u>UninstallLogMode</u>.

Note: Setup can only append to uninstall log files that were created by an Inno Setup 1.3.1 (or later) installation.

Pascal Scripting: Boolean Expressions

See <u>Pascal Scripting: Check Parameters</u> for more information on boolean expressions.

Same Application

"Same application" refers to two separate installations that share the same <u>AppId</u> setting (or if AppId is not set, the same <u>AppName</u> setting), and the same <u>install mode</u> (32-bit or 64-bit).

Pascal Scripting: ExtractTemporaryFile

Prototype:

procedure ExtractTemporaryFile(const FileName: String);

Description:

Extracts the specified file from the [Files] section to a temporary directory. To find the location of the temporary directory, use ExpandConstant('{tmp}').

The extracted files are automatically deleted when Setup exits.

An exception will be raised if the file wasn't extracted successfully, if the file wasn't found, or if the file was found but couldn't be processed because of its MinVersion and/or OnlyBelowVersion parameters.

Remarks:

Use Flags: dontcopy in the [Files] section to tell Setup to skip the file during the normal file copying stage.

Use Flags: noencryption in the [Files] section if encryption is enabled and you call the ExtractTemporaryFile function prior to the user entering the correct password.

When solid compression is enabled, be sure to list your temporary files at (or near) the top of the [Files] section. In order to extract an arbitrary file in a solid-compressed installation, Setup must first decompress all prior files (to a temporary buffer in memory). This can result in a substantial delay if a number of other files are listed above the specified file in the [Files] section.

Example:

```
[Files]
Source: "Readme.txt"; Flags: dontcopy noencryption
[Code]
function InitializeSetup: Boolean;
var
S: AnsiString;
```
```
begin
   // Show the contents of Readme.txt (non Unicode) in
   ExtractTemporaryFile('Readme.txt');
   if LoadStringFromFile(ExpandConstant('{tmp}\Readme.t
    begin
        MsgBox(S, mbInformation, MB_OK);
   end;
```

Result := True; end;

See also:

ExtractTemporaryFiles

Source Directory

By default, the Setup Compiler expects to find files referenced in the script's [Files] section Source parameters, and files referenced in the [Setup] section, under the same directory the script file is located if they do not contain fully qualified pathnames. To specify a different source directory, create a <u>SourceDir</u> directive in the script's [Setup] section.

Pascal Scripting: GetWinDir

Prototype:

function GetWinDir: String;

Description:

Returns fully qualified path of the Windows directory. Only includes a trailing backslash if the Windows directory is the root directory.

User & Group Identifiers

admins	Built-in Administrators group
authusers	Authenticated Users group
everyone	Everyone group
powerusers	Built-in Power Users group
system	Local SYSTEM user
users	Built-in Users group

Pascal Scripting: GetSHA1OfString

Prototype:

```
function GetSHA10fString(const S: AnsiString):
String;
```

Description:

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

Example:

```
var
SHA1: String;
begin
SHA1 := GetSHA10fString('Test');
// SHA1 = '640ab2bae07bedc4c163f679a746f7ab7fb5d1fa'
end;
```

Pascal Scripting: SetPreviousData

Prototype:

function SetPreviousData(const PreviousDataKey: Integer; const ValueName, ValueData: String): Boolean;

Description:

Sets a value that can be restored later using GetPreviousData. Call SetPreviousData inside a RegisterPreviousData event function, once per setting.

Pascal Scripting: RegisterExtraCloseApplicationsResource

Prototype:

function
RegisterExtraCloseApplicationsResource(const
DisableFsRedir: Boolean; const AFilename: String):
Boolean;

Description:

Register an extra file which Setup should check for being in-use. Call RegisterExtraCloseApplicationsResource inside a RegisterExtraCloseApplicationsResources event function, once per file. Returns True if successful.

Pascal Scripting: GetCmdTail

Prototype:
function GetCmdTail: String;

Description:

Returns all command line parameters passed to Setup or Uninstall as a single string.

Pascal Scripting: ParamCount

Prototype:
function ParamCount: Integer;

Description:

Returns the number of command line parameters passed to Setup or Uninstall.

Pascal Scripting: ParamStr

Prototype:

function ParamStr(Index: Integer): String;

Description:

Returns the Index-th command line parameter passed to Setup or Uninstall.

Pascal Scripting: ActiveLanguage

Prototype:
function ActiveLanguage: String;

Description:

Returns the name of the active language.

Pascal Scripting: CustomMessage

Prototype:

```
function CustomMessage(const MsgName: String):
String;
```

Description:

Returns the value of the [CustomMessages] entry with the specified name. If an entry with the specified name does not exist, an exception will be raised.

Example:

```
var
S: String;
begin
S := CustomMessage('CreateDesktopIcon');
// S = 'Create a &desktop icon'
S := FmtMessage(CustomMessage('NameAndVersion'), ['N
// S = 'My Program version 1.0'
```

end;

Pascal Scripting: FmtMessage

Prototype:

```
function FmtMessage(const S: String; const Args:
array of String): String;
```

Description:

Formats the string S using the specified string arguments. A %1 in the format string will be replaced with the first value in the Args array; a %2 will be replaced with the second value; and so on. %% will be replaced with %.

Remarks:

If a %-specifier references a non-existing argument, it will be returned untouched. No exception will be raised.

Example:

```
var
S: String;
begin
S := FmtMessage('%1 version %2 will be installed.',
// S = 'My Program version 1.0 will be installed.'
S := FmtMessage(SetupMessage(msgNotOnThisPlatform),
```

// S = 'This program will not run on Windows 2000.'
end;

Pascal Scripting: SetupMessage

Prototype:

```
function SetupMessage(const ID: TSetupMessageID):
String;
```

Description:

Returns the value of the specified message.

Example:

```
var
S: String;
begin
S := SetupMessage(msgButtonNext);
// S now equals '&Next >'
end;
```

Pascal Scripting: WizardDirValue

Prototype:

function WizardDirValue: String;

Description:

Returns the current contents of the edit control on the *Select Destination Location* page of the wizard.

Unlike ExpandConstant('{app}'), this function will not fail if called after the wizard is shown but prior to the user selecting a directory. Rather, it will return the default directory name.

Pascal Scripting: WizardGroupValue

Prototype:

function WizardGroupValue: String;

Description:

Returns the current contents of the edit control on the *Select Start Menu Folder* page of the wizard.

Unlike ExpandConstant('{group}'), this function will not fail if called after the wizard is shown but prior to the user selecting a folder. Rather, it will return the default folder name.

Pascal Scripting: WizardSetupType

Prototype:

function WizardSetupType(const Description: Boolean): String;

Description:

Returns the name or description of the setup type selected by the user.

Pascal Scripting: WizardSelectedComponents

Prototype:

function WizardSelectedComponents(const Descriptions: Boolean): String;

Description:

Returns a comma-separated list of names or descriptions of the components selected by the user.

Pascal Scripting: WizardSelectedTasks

Prototype:

function WizardSelectedTasks(const Descriptions: Boolean): String;

Description:

Returns a comma-separated list of names or descriptions of the tasks selected by the user.

Pascal Scripting: WizardSilent

Prototype: function WizardSilent: Boolean;

Description:

Returns True if Setup is running silently, False otherwise.

Pascal Scripting: IsUninstaller

Prototype: function IsUninstaller: Boolean;

Description:

Returns True if Uninstall is running as opposed to Setup, False otherwise.

Pascal Scripting: UninstallSilent

Prototype:
function UninstallSilent: Boolean;

Description:

Returns True if Uninstall is running silently, False otherwise.

Pascal Scripting: CurrentFilename

Prototype:

function CurrentFilename: String;

Description:

Returns the destination file name of the [Files] entry that is currently being processed. The returned name may include constants.

Do not attempt to call this function from outside a Check, BeforeInstall or AfterInstall event function belonging to a [Files] entry.

Pascal Scripting: CurrentSourceFilename

Prototype:

function CurrentSourceFilename: String;

Description:

Returns the source file name of the [Files] entry that is currently being processed. The returned name may include constants.

Do not attempt to call this function from outside a Check, BeforeInstall or AfterInstall event function belonging to a [Files] entry with the "external" flag.

Pascal Scripting: ExpandConstant

Prototype:

function ExpandConstant(const S: String): String;

Description:

Changes all constants in S to their values. For example, ExpandConstant('{srcexe}') is changed to the filename of Setup.

An exception will be raised if there was an error expanding the constants.

Pascal Scripting: ExpandConstantEx

Prototype:

function ExpandConstantEx(const S: String; const CustomConst, CustomValue: String): String;

Description:

Changes all constants in S to their values. Additionally, any constant equal to CustomConst will be changed to CustomValue.

An exception will be raised if there was an error expanding the constants.

Pascal Scripting: IsComponentSelected

Prototype:

function IsComponentSelected(const Components: String): Boolean;

Description:

Returns True if the specified component is selected. Multiple components may be specified in the same manner as in a <u>Components parameter</u>.

Example:

begin

```
if IsComponentSelected('helpfiles') then
```

```
// the 'helpfiles' component is selected
```

end;

Pascal Scripting: IsTaskSelected

Prototype:

```
function IsTaskSelected(const Tasks: String):
Boolean;
```

Description:

Returns True if the specified task is selected. Multiple tasks may be specified in the same manner as in a <u>Tasks parameter</u>.

Example:

begin

```
if IsTaskSelected('desktopicon') then
```

```
// the 'desktopicon' task is selected
```

end;

Pascal Scripting: ExtractTemporaryFiles

Prototype:

function ExtractTemporaryFiles(const Pattern: String): Integer;

Description:

Extracts the files matching the wildcard specified by Pattern from the [Files] section to a temporary directory. Returns the number of extracted files. To find the location of the temporary directory, use ExpandConstant('{tmp}').

The extracted files are automatically deleted when Setup exits.

An exception will be raised if no files were extracted successfully, no files were found, or if files were found but none could be processed because of their MinVersion and/or OnlyBelowVersion parameters.

Remarks:

Use Flags: dontcopy in the [Files] section to tell Setup to skip the file during the normal file copying stage.

When solid compression is enabled, be sure to list your temporary files at (or near) the top of the [Files] section. In order to extract an arbitrary file in a solid-compressed installation, Setup must first decompress all prior files (to a temporary buffer in memory). This can result in a substantial delay if a number of other files are listed above the specified file in the [Files] section.

Example:

```
[Files]
Source: "Readme.txt"; Flags: dontcopy
Source: "MyProg.exe"; DestDir: "{app}"
Source: "MyProg.chm"; DestDir: "{app}"
[Code]
function InitializeSetup: Boolean;
var
S: AnsiString;
ResultCode: Integer;
```

begin
 // Show the contents of Readme.txt (non Unicode) in
 ExtractTemporaryFiles('{tmp}\Readme.txt');
 if LoadStringFromFile(ExpandConstant('{tmp}\Readme.t
 begin
 MsgBox(S, mbInformation, MB_OK);
 end;
 // Extract all MyBrog files and launch it. Note how

// Extract all MyProg files and launch it. Note how
ExtractTemporaryFiles('{app}\MyProg.*');
ExecAsOriginalUser(ExpandConstant('{tmp}\')+'{app}\N
SW SHOWNORMAL, ewWaitUntilTerminated, ResultCode);

Result := True; end;

See also: ExtractTemporaryFile

Pascal Scripting: GetPreviousData

Prototype:

function GetPreviousData(const ValueName, DefaultValueData: String): String;

Description:

Gets a value that was previously stored using SetPreviousData.

Pascal Scripting: Terminated

Prototype: function Terminated: Boolean;

Description:

Returns True if Setup or Uninstall is terminating, False otherwise.

Pascal Scripting: RmSessionStarted

Prototype:

function RmSessionStarted: Boolean;

Description:

Returns True if a Restart Manager session was started, False otherwise.

See also:

CloseApplications

Pascal Scripting: Abort

Prototype:

procedure Abort;

Description:

Escapes from the current execution path without reporting an error.

Abort raises a special "silent exception" which operates like any other exception, but does not display an error message to the end user.

Remarks:

Abort does not cause Setup or Uninstall to exit unless it's called from one of these event functions (or another function invoked by them):

InitializeSetup
InitializeWizard
CurStepChanged(ssInstall)

InitializeUninstall
CurUninstallStepChanged(usAppMutexCheck)
CurUninstallStepChanged(usUninstall)

See also: PrepareToInstall

Pascal Scripting: RaiseException

Prototype:

procedure RaiseException(const Msg: String);

Description:

Raises an exception with the specified message.

Example:

begin

RaiseException('Your message goes here');

// The following line will not be executed because (
 MsgBox('You will not see this.', mbInformation, MB_(
 end;

Pascal Scripting: GetExceptionMessage

Prototype:

function GetExceptionMessage: String;

Description:

Returns the message associated with the current exception. This function should only be called from within an except section, or a function called from an except section.

Remarks:

Exception messages generally do not end in a period. Pass the result of this function to AddPeriod to add one.

Example:

```
var

I: Integer;

begin

I := 1;

try

// The following line will raise a "Division by ze

I := I div 0;

except

// Catch the exception, deal with it, and continue

MsgBox('We caught this exception: ' + AddPeriod(Ge

mbError, MB_OK);

end;

end;
```
Pascal Scripting: ShowExceptionMessage

Prototype:

procedure ShowExceptionMessage;

Description:

Shows the message associated with the current exception in a message box. This function should only be called from within an except section, or a function called from an except section.

Remarks:

If logging is enabled (via the <u>/LOG</u> command line parameter or the <u>SetupLogging</u> [Setup] section directive) the message will recorded in the log in addition to being shown.

Example:

```
var

I: Integer;

begin

I := 1;

try

// The following line will raise a "Division by ze

I := I div 0;

except

// Catch the exception, show it, and continue

ShowExceptionMessage;

end;

end;
```

Pascal Scripting: IsAdminLoggedOn

Prototype:

function IsAdminLoggedOn: Boolean;

Description:

Returns True if the user account that Setup/Uninstall is running under is a member of the local Administrators group.

Pascal Scripting: IsPowerUserLoggedOn

Prototype:

function IsPowerUserLoggedOn: Boolean;

Description:

Returns True if the user account that Setup/Uninstall is running under is a member of the Power Users group.

Pascal Scripting: GetWindowsVersion

Prototype:

function GetWindowsVersion: Cardinal;

Description:

Returns the version number of Windows packed into a single integer. The upper 8 bits specify the major version; the following 8 bits specify the minor version; the lower 16 bits specify the build number. For example, this function will return \$05000893 on Windows 2000, which is version 5.0.2195.

To retrieve just the major version number, use: "GetWindowsVersion shr 24". To retrieve just the minor version number, use: " (GetWindowsVersion shr 16) and \$FF". To retrieve just the build number, use: "GetWindowsVersion and \$FFFF".

Example:

```
function IsWindowsXPOrLater: Boolean;
begin
    Result := (GetWindowsVersion >= $05010000);
end;
```

See also:

<u>GetWindowsVersionEx</u>

Pascal Scripting: GetWindowsVersionEx

Prototype:

procedure GetWindowsVersionEx(var Version: TWindowsVersion);

Description:

Returns extended information about the version of Windows in a record.

TWindowsVersion is defined as:

TWindowsVersion = record	
Major: Cardinal;	<pre>// Major version numt</pre>
Minor: Cardinal;	<pre>// Minor version numk</pre>
Build: Cardinal;	// Build number
ServicePackMajor: Cardinal;	// Major version numk
ServicePackMinor: Cardinal;	<pre>// Minor version numk</pre>
NTPlatform: Boolean;	// True if an NT-bas€
<pre>ProductType: Byte;</pre>	<pre>// Product type (see</pre>
SuiteMask: Word;	// Product suites in:
end;	

The ProductType field can be one of the following values:

VER_NT_WORKSTATION VER_NT_DOMAIN_CONTROLLER VER_NT_SERVER

It can also be zero if the product type could not be determined (unlikely). VER_NT_WORKSTATION indicates a non-server edition of Windows (e.g. Workstation, Professional, or Home).

The SuiteMask field can be a combination of the following values:

VER_SUITE_BACKOFFICE VER_SUITE_BLADE VER_SUITE_DATACENTER VER_SUITE_ENTERPRISE VER_SUITE_EMBEDDEDNT VER_SUITE_PERSONAL

```
VER_SUITE_SINGLEUSERTS
VER_SUITE_SMALLBUSINESS
VER_SUITE_SMALLBUSINESS_RESTRICTED
VER_SUITE_TERMINAL
```

VER_SUITE_PERSONAL, for example, is set on Home edition of Windows XP, and VER_SUITE_BLADE is set on the Web edition of Windows Server 2003.

Example:

The following example demonstrates how you can disallow installation on certain editions of Windows, and check service pack levels on multiple operating system versions. (Neither of these things are possible with the MinVersion [Setup] section directive.)

```
function InitializeSetup: Boolean;
var
 Version: TWindowsVersion;
 S: String;
begin
 GetWindowsVersionEx(Version);
  // Disallow installation on Home edition of Windows
  if Version.SuiteMask and VER SUITE PERSONAL <> 0 the
  begin
    SuppressibleMsgBox('This program cannot be instal]
      mbCriticalError, MB_OK, IDOK);
    Result := False;
    Exit;
  end;
  // Disallow installation on domain controllers
  if Version.ProductType = VER_NT_DOMAIN_CONTROLLER th
  begin
    SuppressibleMsgBox('This program cannot be instal]
      mbCriticalError, MB OK, IDOK);
    Result := False;
    Exit;
```

end;

```
// On Windows 2000, check for SP4
  if Version.NTPlatform and
     (Version.Major = 5) and
     (Version.Minor = 0) and
     (Version.ServicePackMajor < 4) then
  begin
    SuppressibleMsgBox('When running on Windows 2000,
      mbCriticalError, MB_OK, IDOK);
    Result := False;
    Exit;
  end;
  // On Windows XP, check for SP2
  if Version.NTPlatform and
     (Version.Major = 5) and
     (Version.Minor = 1) and
     (Version.ServicePackMajor < 2) then
  begin
    SuppressibleMsgBox('When running on Windows XP, S\epsilon
      mbCriticalError, MB OK, IDOK);
    Result := False;
    Exit;
  end;
 Result := True;
end;
```

Pascal Scripting: GetWindowsVersionString

Prototype:

function GetWindowsVersionString: String;

Description:

Returns the version number of Windows in string form. On Windows 2000, for example, this function will return "5.00.2195".

Pascal Scripting: IsWin64

Prototype:

function IsWin64: Boolean;

Description:

Returns True if the system is running a 64-bit version of Windows that provides the API support Inno Setup requires to perform 64-bit installation tasks. If False is returned, you cannot utilize any of Inno Setup's 64-bit-only features.

Remarks:

This function will always return True on an x64 edition of Windows.

For this function to return True on an Itanium edition of Windows, the system must be running Windows Server 2003 SP1 or later. Older versions lack APIs that Inno Setup requires (e.g. RegDeleteKeyEx).

Example:

begin

```
// Check IsWin64 before using a 64-bit-only feature
// avoid an exception when running on 32-bit Windows
if IsWin64 then
begin
    MsgBox('64-bit program files reside in: ' +
    ExpandConstant('{pf64}'), mbInformation, MB_OK);
end;
end;
```

See also:

Is64BitInstallMode ProcessorArchitecture

Pascal Scripting: Is64BitInstallMode

Prototype:

function Is64BitInstallMode: Boolean;

Description:

Returns True if Setup or Uninstall is running in <u>64-bit mode</u>, or False if it is running in <u>32-bit mode</u>.

Remarks:

When True is returned, it is safe to assume that <u>IsWin64</u> will also return True.

Example:

begin

if Is64BitInstallMode then

MsgBox('Installing in 64-bit mode', mbInformation, else

MsgBox('Installing in 32-bit mode', mbInformation, end;

See also:

IsWin64 ProcessorArchitecture

Pascal Scripting: ProcessorArchitecture

Prototype:

function ProcessorArchitecture: TSetupProcessorArchitecture;

Description:

Returns the native processor architecture of the current system.

TSetupProcessorArchitecture is defined as:

```
TSetupProcessorArchitecture = (paUnknown, paX86,
paX64, paIA64);
```

Remarks:

A 64-bit processor architecture will never be returned on 32-bit versions of Windows. Hence, you cannot use this function to detect a 64-bit AMD CPU on a 32-bit version of Windows; you'll just get back paX86 if you try.

paUnknown is returned if Setup/Uninstall does not recognize the processor architecture. It can be assumed that an "unknown" architecture is at least capable of executing 32-bit code, or Setup/Uninstall wouldn't be running at all.

If paIA64 is returned, and ia64 is *not* included in the value of the <u>ArchitecturesInstallIn64BitMode</u> [Setup] section directive, you should not assume that Inno Setup's 64-bit-only features are available -- for example, the {pf64} constant. Those features only work when <u>IsWin64</u> returns True, and as documented, it may not return True on older Itanium versions of Windows that lack certain APIs Inno Setup requires.

Therefore, instead of:

```
if ProcessorArchitecture = paIA64 then
```

```
// perform some Itanium-specific install task that
```

```
// involves expanding {pf64}
```

you should additionally check that IsWin64 returns True:

```
if ProcessorArchitecture = paIA64 then
```

```
begin
  if IsWin64 then
    // perform some Itanium-specific install task th
    // involves expanding {pf64}
    else
        // cannot use 64-bit features; display an error
        // fail silently, try something else, etc.
end;
```

If ia64 *is* included in the value of the <u>ArchitecturesInstallIn64BitMode</u> [Setup] section directive, then it is not necessary to check IsWin64 because Setup will do so itself at startup, and fail with an error message (MissingW0W64APIs) if it is False.

Example:

```
var
S: String;
begin
case ProcessorArchitecture of
paX86: S := 'x86';
paX64: S := 'x64';
paIA64: S := 'Itanium';
else
S := 'Unrecognized';
end;
MsgBox('Your processor architecture: ' + S, mbInform
end;
```

See also:

IsWin64 Is64BitInstallMode

Pascal Scripting: InstallOnThisVersion

Prototype:

function InstallOnThisVersion(const MinVersion, OnlyBelowVersion: String): Boolean;

Description:

This function is deprecated. Returns True if an entry with the specified MinVersion and OnlyBelowVersion parameters should be installed. If an invalid version string is passed, an exception will be raised.

This function is provided for backward compatibility only, and may be removed in a future release. New scripts should use <u>GetWindowsVersion</u> or <u>GetWindowsVersionEx</u> instead.

Remarks:

Prior to Inno Setup 5.5.0, this function returned irInstall rather than a Boolean True value. irInstall is now defined as an alias for True.

Example:

```
// Old method, deprecated
function IsWindowsXPOrLater: Boolean;
begin
   Result := InstallOnThisVersion('0,5.1', '0,0');
end;
```

// New method

function IsWindowsXPOrLater: Boolean;

begin

```
Result := (GetWindowsVersion >= $05010000);
end;
```

See also:

GetWindowsVersion GetWindowsVersionEx

Pascal Scripting: GetEnv

Prototype:

function GetEnv(const EnvVar: String): String;

Description:

Gets the value of the specified environment variable.

Pascal Scripting: GetUserNameString

Prototype:

function GetUserNameString: String;

Description:

Retrieves the name of the user currently logged onto the system.

Pascal Scripting: GetComputerNameString

Prototype:

function GetComputerNameString: String;

Description:

Retrieves the name of the computer the Setup or Uninstall program is running on (as returned by the Windows *GetComputerName* function).

Pascal Scripting: GetUILanguage

Prototype:

function GetUILanguage: Integer;

Description:

Returns the language identifier (LANGID) of the current user's UI language, which is either the language of Windows itself, or in the case of a MUI edition of Windows, the user interface language chosen in Control Panel's Regional Options. Returns 0 if the function fails (unlikely).

Remarks:

Refer to the list of valid language identifiers on MSDN .

Example:

begin

```
if GetUILanguage = $0409 then
begin
   // UI language is English (United States)
end;
// You can use "and $3FF" to extract the primary lar
if GetUILanguage and $3FF = $09 then
begin
   // Matches any variant of English
end;
```

end;

Pascal Scripting: FontExists

Prototype:

function FontExists(const FaceName: String):
Boolean;

Description:

Returns True if a font with the specified face name is installed on the system.

Pascal Scripting: FindWindowByClassName

Prototype:

function FindWindowByClassName(const ClassName: String): HWND;

Description:

Retrieves a handle to the top-level window whose class name matches the specified string. This function does not search child windows, and does not perform a case-sensitive search. Returns 0 if no window is found.

Pascal Scripting: FindWindowByWindowName

Prototype:

function FindWindowByWindowName(const WindowName: String): HWND;

Description:

Retrieves a handle to the top-level window whose window name matches the specified string. This function does not search child windows, and does not perform a case-sensitive search. Returns 0 if no window is found.

Pascal Scripting: SendMessage

Prototype:

function SendMessage(const Wnd: HWND; const Msg, WParam, LParam: Longint): Longint;

Description:

Sends the specified message to the specified window. Does not return until the window procedure has processed the message.

Pascal Scripting: PostMessage

Prototype:

function PostMessage(const Wnd: HWND; const Msg, WParam, LParam: Longint): Boolean;

Description:

Posts the specified message to the specified window, returning immediately. Returns True if successful.

Pascal Scripting: SendNotifyMessage

Prototype:

function SendNotifyMessage(const Wnd: HWND; const Msg, WParam, LParam: Longint): Boolean;

Description:

Sends the specified message to the specified window without waiting for the message to be processed by the destination window procedure. Returns True if successful.

Pascal Scripting: RegisterWindowMessage

Prototype:

function RegisterWindowMessage(const Name: String):
Longint;

Description:

The RegisterWindowMessage function defines a new window message that is guaranteed to be unique throughout the system. The returned message value can be used when calling the SendBroadcastMessage or PostBroadcastMessage function.

Pascal Scripting: SendBroadcastMessage

Prototype:

function SendBroadcastMessage(const Msg, WParam, LParam: Longint): Longint;

Description:

Sends the specified message to top-level windows in the system. Does not return until all window procedure have processed the message. The specified message must be unique. Use RegisterWindowMessage to get such a message.

Pascal Scripting: PostBroadcastMessage

Prototype:

function PostBroadcastMessage(const Msg, WParam, LParam: Longint): Boolean;

Description:

Posts the specified message to top-level windows in the system, returning immediately.

The specified message must be unique. Use RegisterWindowMessage to get such a message.

Pascal Scripting: SendBroadcastNotifyMessage

Prototype:

function SendBroadcastNotifyMessage(const Msg, WParam, LParam: Longint): Boolean;

Description:

not yet available

Pascal Scripting: CreateMutex

Prototype:

procedure CreateMutex(const Name: String);

Description:

Creates a mutex with the specified name.

Pascal Scripting: CheckForMutexes

Prototype:

function CheckForMutexes(Mutexes: String): Boolean;

Description:

Returns True if any of the mutexes in the comma-separated Mutexes string exist.

Pascal Scripting: MakePendingFileRenameOperationsChecksum

Prototype:

procedure MakePendingFileRenameOperationsChecksum: String;

Description:

Calculates a checksum of the current PendingFileRenameOperations registry value. The caller can use this checksum to determine if PendingFileRenameOperations or WININIT.INI was changed (perhaps by another program).

Example:

var

ChecksumBefore, ChecksumAfter: String;

begin

ChecksumBefore := MakePendingFileRenameOperationsChe
// ...run a program...

ChecksumAfter := MakePendingFileRenameOperationsChec
if ChecksumAfter <> ChecksumBefore then

// PendingFileRenameOperations or WININIT.INI char end;

Pascal Scripting: UnloadDLL

Prototype:

procedure UnloadDLL(Filename: String);

Description:

Unloads the specified DLL that was loaded by the [Code] section using an "external" keyword. This can be useful if you need to delete the DLL.

The case of the filename and any path name must exactly match that of the function import. You will need to expand any constants in the filename yourself before passing it to UnloadDLL.

If the function import used a "files:" prefix, prepend the value of the $\{tmp\}$ constant to the filename (e.g.

ExpandConstant('{tmp}\filename.dll')).

Remarks:

It's not recommended that you try this, but if you attempt to call a function in a DLL that has been unloaded, the DLL will be re-loaded.

Example:

procedure DllFunc; external 'DllFunc@{app}\MyDll.dll

. . .

begin // Call DllFunc DllFunc;

// Unload the DLL
UnloadDLL(ExpandConstant('{app}\MyDll.dll'));

```
// Now we can delete the DLL
DeleteFile(ExpandConstant('{app}\MyDll.dll'));
end;
```

Pascal Scripting: DLLGetLastError

Prototype:

function DLLGetLastError(): Longint;

Description:

Returns value the last error code had right after the most recent DLL function call you made. Useful after calling Windows API functions (if the function sets the last error code).

Remarks:

It's recommended to use this function instead of directly calling the GetLastError Windows API function since Setup or Uninstall makes API calls of its own, so the last error code could be overwritten at any time.

Refer to the system error codes on MSDN .

Example:

```
function MessageBox(hWnd: Integer; lpText, lpCaption:
```

• • •

```
begin
if MessageBox(-1, '', '', -1) = 0 then
MsgBox(SysErrorMessage(DLLGetLastError), mbError,
```

Pascal Scripting: Chr

Prototype: function Chr(B: Byte): Char;

Description:

Returns the character with the specified ordinal value.

Pascal Scripting: Ord

Prototype:
function Ord(C: Char): Byte;

Description:

Returns the ordinal value of the specified character.

Pascal Scripting: Copy

Prototype:

function Copy(S: String; Index, Count: Integer):
String;

Description:

Returns a string containing Count characters starting at S[Index]. If Index is larger than the length of S, Copy returns an empty string. If Count specifies more characters than are available, only the characters from S[Index] to the end of S are returned.

Pascal Scripting: Length

Prototype: function Length(s: String): Longint;

Description:

Returns the length of the specified string.
Pascal Scripting: Lowercase

Prototype:

function Lowercase(S: String): String;

Description:

Returns a copy of the string S, but with all 7-bit ASCII characters between 'A' and 'Z' converted to lowercase. To convert 8-bit international characters, use AnsiLowercase instead.

Pascal Scripting: Uppercase

Prototype:

function Uppercase(S: String): String;

Description:

Returns a copy of the string S, but with all 7-bit ASCII characters between 'a' and 'z' converted to uppercase. To convert 8-bit international characters, use AnsiUppercase instead.

Pascal Scripting: AnsiLowercase

Prototype:

function AnsiLowercase(S: String): String;

Description:

Returns a string that is a copy of the given string converted to lowercase. The conversion uses the current Windows locale. This function supports multi-byte character sets (MBCS).

Pascal Scripting: AnsiUppercase

Prototype:

function AnsiUppercase(S: String): String;

Description:

Returns a string that is a copy of the given string converted to uppercase. The conversion uses the current Windows locale. This function supports multi-byte character sets (MBCS).

Pascal Scripting: StringOfChar

Prototype:

function StringOfChar(c: Char; I : Longint): String;

Description:

Returns a string of length I with all characters set to character C.

Pascal Scripting: Delete

Prototype:

procedure Delete(var S: String; Index, Count: Integer);

Description:

Removes a substring of Count characters from string S starting at S[Index].

If Index is larger than the length of S, no characters are deleted. If Count specifies more characters than remain starting at the S[Index], Delete removes the rest of the string.

Pascal Scripting: Insert

Prototype:

procedure Insert(Source: String; var Dest: String; Index: Integer);

Description:

Merges Source into Dest at the position Dest[Index].

Pascal Scripting: StringChange

Prototype:

function StringChange(var S: String; const FromStr, ToStr: String): Integer;

Description:

This function is deprecated. It is equivalent to calling <u>StringChangeEx</u> with the SupportMBCS parameter set to False.

Pascal Scripting: StringChangeEx

Prototype:

function StringChangeEx(var S: String; const
FromStr, ToStr: String; const SupportDBCS: Boolean):
Integer;

Description:

Changes all occurrences in S of FromStr to ToStr. If SupportDBCS is True (recommended unless you require binary safety), double-byte character sequences in S are recognized and handled properly. Otherwise, the function behaves in a binary-safe manner. Returns the number of times FromStr was matched and changed.

Remarks:

When working with strings containing paths or filenames, be sure to pass True in the SupportDBCS parameter. Otherwise, paths with Chinese, Japanese, or Korean characters may be corrupted.

Example:

```
var
S: String;
begin
S := ExpandConstant('{commonappdata}');
// S = 'C:\Documents and Settings\All Users\Applicat
StringChangeEx(S, '\', '/', True);
// S = 'C:/Documents and Settings/All Users/Applicat
end;
```

Pascal Scripting: Pos

Prototype:

function Pos(SubStr, S: String): Integer;

Description:

Searches for Substr within S and returns an integer value that is the index of the first character of Substr within S. If Substr is not found, Pos returns zero.

Pascal Scripting: AddQuotes

Prototype:

function AddQuotes(const S: String): String;

Description:

Adds a quote (") character to the left and right sides of the string if the string contains a space and it didn't have quotes already. This is primarily used when spawning another process with a long filename as one of the parameters.

Pascal Scripting: RemoveQuotes

Prototype:

function RemoveQuotes(const S: String): String;

Description:

Opposite of AddQuotes; removes any quotes around the string.

Pascal Scripting: ConvertPercentStr

Prototype:

function ConvertPercentStr(var S: String): Boolean;

Description:

Expands all %-encoded characters in the string (see <u>RFC 2396</u>). Returns True if all were successfully expanded.

Pascal Scripting: CompareText

Prototype:

function CompareText(const S1, S2: string): Integer;

Description:

Compares the strings S1 and S2 and returns 0 if they are equal. If S1 is greater than S2, CompareText returns an integer greater than 0. If S1 is less than S2, CompareText returns an integer less than 0. The CompareText function is not case sensitive.

Pascal Scripting: CompareStr

Prototype:

function CompareStr(const S1, S2: string): Integer;

Description:

Compares S1 to S2, with case-sensitivity. The return value is less than 0 if S1 is less than S2, 0 if S1 equals S2, or greater than 0 if S1 is greater than S2.

Pascal Scripting: Format

Prototype:

```
function Format(const Format: string; const Args:
array of const): string;
```

Description:

Formats the series of arguments in the open array Args. Formatting is controlled by the format string Format; the results are returned as a string.

An exception will be raised if an invalid format string is specified, too few arguments are passed, or if any arguments are of the wrong type.

Example:

```
var

S: String;

I: Integer;

begin

S := Format('%d files found', [10]);

// S = '10 files found'

S := Format('Filename: %s', ['file.txt']);

// S = 'Filename: file.txt'

I := 64;

S := Format('%d in hex, padded to 8 digits: %.8x', |

// S = '64 in hex, padded to 8 digits: 00000040'

end;
```

Pascal Scripting: Trim

Prototype:

function Trim(const S: string): String;

Description:

Trims leading and trailing spaces and control characters from the given string S.

Pascal Scripting: TrimLeft

Prototype: function TrimLeft(const S: string): String;

Description:

Trims leading spaces and control characters from the given string S.

Pascal Scripting: TrimRight

Prototype: function TrimRight(const S: string): String;

Description:

Trims trailing spaces and control characters from the given string S.

Pascal Scripting: StrToIntDef

Prototype:

```
function StrToIntDef(s: string; def: Longint):
Longint;
```

Description:

The StrToInt function converts the string passed in S into a number. If S does not represent a valid number, StrToInt returns the number passed in Def.

Pascal Scripting: StrToInt

Prototype:

function StrToInt(s: string): Longint;

Description:

The StrToInt function converts the string passed in S into a number.

Remarks:

Use of <u>StrToIntDef</u> instead of StrToInt is recommended.

Pascal Scripting: StrToInt64Def

Prototype:

function StrToInt64Def(s: string; def: Int64): Int64;

Description:

The StrToInt64Def function converts the string passed in S into a 64-bit number. If S does not represent a valid 64-bit number, StrToInt returns the 64-bit number passed in Def.

Requires Unicode Inno Setup.

Pascal Scripting: StrToInt64

Prototype:

function StrToInt64(s: string): Int64;

Description:

The StrToInt64 function converts the string passed in S into a 64-bit number.

Requires Unicode Inno Setup.

Remarks:

Use of <u>StrToInt64Def</u> instead of StrToInt64 is recommended.

Pascal Scripting: StrToFloat

Prototype:

function StrToFloat(s: string): Extended;

Description:

The StrToFloat function converts the string passed in S into a floating point number. The character used for the decimal point should always be a dot.

Pascal Scripting: IntToStr

Prototype:

function IntToStr(i: Int64): String;

Description:

The IntToStr function converts an 32-bit or 64-bit number into a string.

64-bit numbers require Unicode Inno Setup.

Pascal Scripting: FloatToStr

Prototype:

function FloatToStr(e: extended): String;

Description:

The FloatToStr function converts a floating point number into a string.

Unicode Inno Setup: the character used for the decimal point will always be a dot. Non-Unicode Inno Setup: the character used for the decimal point is locale-dependant.

Pascal Scripting: CharLength

Prototype:

function CharLength(const S: String; const Index: Integer): Integer;

Description:

Returns the length in bytes (1 or 2) of the character in the specified string at the specified index.

Remarks:

In double-byte character sets (Chinese, Japanese, Korean), most non-ASCII characters occupy two bytes. Note that the second byte of a double-byte character -- known as the "trail byte" -- can be in the same range used by ASCII characters (below 128). Thus, when stepping through a string that may contain double-byte characters, such as a path or filename, care must be taken to not mistake trail bytes for single-byte ASCII characters.

Example:

```
function BackslashToSlash(const S: String): String;
var
    I: Integer;
begin
    Result := S;
    I := 1;
    while I <= Length(Result) do
    begin
        if Result[I] = '\' then
            Result[I] := '/';
        // Go to the next character. But do not simply inc
        // Increment by CharLength() in case Result[I] is
        I := I + CharLength(Result, I);
    end;
end;
```

begin

// Show path of Common Files with backslashes change
MsgBox(BackslashToSlash(ExpandConstant('{cf}')), mb]
end;

Pascal Scripting: AddBackslash

Prototype:

function AddBackslash(const S: String): String;

Description:

Returns the specified string with a trailing backslash added, unless the string is empty or already ends in a slash or backslash.

Pascal Scripting: RemoveBackslashUnlessRoot

Prototype:

function RemoveBackslashUnlessRoot(const S: String):
String;

Description:

Returns the specified string with any trailing slashes/backslashes removed, unless the string specifies the root directory of a drive (e.g. "C:\" or "\"), in which case it leaves 1 slash.

Pascal Scripting: RemoveBackslash

Prototype:

function RemoveBackslash(const S: String): String;

Description:

Returns the specified string with any trailing slashes/backslashes removed.

Remarks:

Use of this function is discouraged; use <u>RemoveBackslashUnlessRoot</u> instead when working with file system paths.

Pascal Scripting: AddPeriod

Prototype:

function AddPeriod(const S: String): String;

Description:

Returns the specified string with a trailing period added, unless the string is empty or already ends in a period or other punctuation mark.

Pascal Scripting: ChangeFileExt

Prototype:

function ChangeFileExt(const FileName, Extension: string): String;

Description:

Takes the file name passed in FileName and changes the extension of the file name to the extension passed in Extension.

Pascal Scripting: ExtractFileExt

Prototype:

function ExtractFileExt(const FileName: string):
String;

Description:

Extracts the extension part of the given file name. The resulting string includes the period character that separates the name and extension parts. The resulting string is empty if the given filename has no extension.

Pascal Scripting: ExtractFileDir

Prototype:

function ExtractFileDir(const FileName: string):
String;

Description:

Extracts the drive and directory parts of the given file name. The resulting string is empty if FileName contains no drive and directory parts.

Pascal Scripting: ExtractFilePath

Prototype:

function ExtractFilePath(const FileName: string):
String;

Description:

Extracts the drive and directory parts of the given file name. The resulting string is the leftmost characters of FileName, up to and including the colon or backslash that separates the path information from the name and extension. The resulting string is empty if FileName contains no drive and directory parts.
Pascal Scripting: ExtractFileName

Prototype:

function ExtractFileName(const FileName: string):
String;

Description:

Extracts the name and extension parts of the given file name. The resulting string is the rightmost characters of FileName, starting with the first character after the colon or backslash that separates the path information from the name and extension. The resulting string is equal to FileName if FileName contains no drive and directory parts.

Pascal Scripting: ExtractFileDrive

Prototype:

function ExtractFileDrive(const FileName: string):
String;

Description:

Returns a string containing the 'drive' portion of a fully qualified path name for the file passed in the FileName. For file names with drive letters, the resulting string is in the form '<drive>:'. For file names with a UNC path the resulting string is in the form '\\<servername>\ <sharename>'. If the given path contains neither style of path prefix, the result is an empty string.

Pascal Scripting: ExtractRelativePath

Prototype:

function ExtractRelativePath(const BaseName, DestName: String): String;

Description:

Converts a fully qualified path name into a relative path name. The DestName parameter specifies the file name (including path) to be converted. BaseName is the fully qualified name of the base directory to which the returned path name should be relative. BaseName may or may not include a file name, but it must include the final path delimiter.

ExtractRelativePath strips out common path directories and inserts '..\' for each level up from the BaseName.

```
var
S: String;
begin
S := ExtractRelativePath('c:\windows\system32\', 'c:
// S = ..\..\autoexec.bat
end;
```

Pascal Scripting: ExpandFileName

Prototype:

function ExpandFileName(const FileName: string):
String;

Description:

Returns a string containing a fully qualified path name for the file passed in the FileName. A fully qualified path name includes the drive letter and any directory and subdirectories in addition to the file name and extension.

Pascal Scripting: ExpandUNCFileName

Prototype:

function ExpandUNCFileName(const FileName: string):
String;

Description:

Returns a string containing a fully qualified path name for the file passed in the FileName. A fully qualified path name includes the drive portion of the filename in the UNC format '\\<servername>\ <sharename>' if the drive letter is mapped to a network resource instead of a local drive and any directory and subdirectories in addition to the file name and extension.

Pascal Scripting: GetDateTimeString

Prototype:

function GetDateTimeString(const DateTimeFormat: String; const DateSeparator, TimeSeparator: Char): String;

Description:

Returns the current date and time as a string using the specified formatting. 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. dddddd 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). vy 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 #0, 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 #0, 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.

```
GetDateTimeString('ddddd', #0, #0);
GetDateTimeString('ddddd tt', #0, #0);
GetDateTimeString('dd/mm/yyyy hh:nn:ss', '-', ':');
```

Pascal Scripting: SetLength

Prototype:

procedure SetLength(var S: String; L: Longint);

Description:

Sets the length of a string.

Pascal Scripting: CharToOemBuff

Prototype:

procedure CharToOemBuff(var S: AnsiString);

Description:

Translates an ANSI string to a string with characters from the OEMdefined character set.

Pascal Scripting: OemToCharBuff

Prototype:

procedure OemToCharBuff(var S: AnsiString);

Description:

Translates a string with characters from the OEM-defined character set into an ANSI string.

Pascal Scripting: GetMD5OfString

Prototype:

```
function GetMD50fString(const S: AnsiString):
String;
```

Description:

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

```
var
MD5: String;
begin
MD5 := GetMD50fString('Test');
// MD5 = '0cbc6611f5540bd0809a388dc95a615b'
end;
```

Pascal Scripting: GetMD5OfUnicodeString

Prototype:

```
function GetMD50fUnicodeString(const S: String):
String;
```

Description:

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

Causes an internal error if called by non Unicode Setup or Uninstall.

```
var
MD5: String;
begin
MD5 := GetMD50fUnicodeString('Test');
// MD5 = '8e06915d5f5d4f8754f51892d884c477'
end;
```

Pascal Scripting: GetSHA1OfUnicodeString

Prototype:

```
function GetSHA10fUnicodeString(const S: String):
String;
```

Description:

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

Causes an internal error if called by non Unicode Setup or Uninstall.

```
var
SHA1: String;
begin
SHA1 := GetSHA10fUnicodeString('Test');
// SHA1 = '9ab696a37604d665dc97134dbee44cfe70451b1a'
end;
```

Pascal Scripting: SysErrorMessage

Prototype:

function SysErrorMessage(ErrorCode: Integer):
String;

Description:

Returns a localized error message string that corresponds to the given operating system error code.

Remarks:

Refer to the system error codes on MSDN .

Pascal Scripting: MinimizePathName

Prototype:

function MinimizePathName(const Filename: String; const Font: TFont; MaxLen: Integer): String;

Description:

Returns a minimized filename that will not take more than MaxLen pixels to display with the given font. The minimizing is done by replacing the middle part with '...' as needed.

Example:

MyLabel.Caption := MinimizePathName(MyPathName, MyLabe

See also:

<u>TFont</u>

Pascal Scripting: GetArrayLength

Prototype:

function GetArrayLength(var Arr: Array): Longint;

Description:

Gets the length of an array.

Pascal Scripting: SetArrayLength

Prototype:

procedure SetArrayLength(var Arr: Array; I: Longint);

Description:

Sets the length of an array. Always call SetArrayLength before accessing the elements in an array.

Pascal Scripting: Null

Prototype:
function Null: Variant;

Description:

Returns a variant with a value of Null.

Pascal Scripting: Unassigned

Prototype: function Unassigned: Variant;

Description:

Returns a variant with no value assigned.

Pascal Scripting: VarIsEmpty

Prototype:

function VarIsEmpty(const V: Variant): Boolean;

Description:

Non Unicode Inno Setup: Returns True if the specified variant's value is undefined. Unicode Inno Setup: Returns True if the specified variant is unassigned.

See also:

<u>VarIsClear</u>

Pascal Scripting: VarIsClear

Prototype:

function VarIsClear(const V: Variant): Boolean;

Description:

Returns True if the specified variant's value is undefined.

Remarks:

Requires Unicode Inno Setup.

See also:

<u>VarIsEmpty</u>

Pascal Scripting: VarlsNull

Prototype:

function VarIsNull(const V: Variant): Boolean;

Description:

Returns True if the specified variant has a value of Null assigned.

Pascal Scripting: VarType

Prototype:

function VarType(const V: Variant): TVarType;

Description:

Returns the type code of the specified variant.

The lower twelve bits of a variant type code (the bits defined by the varTypeMask bit mask) define the type of the variant. The varArray bit is set if the variant is an array of the given type. The varByRef bit is set if the variant is a reference to a value of the given type as opposed to an actual value.

The following variant type code constants are defined:

const

varEmpty	=	\$0000;
varNull	=	\$0001;
varSmallint	=	\$0002;
varInteger	=	\$0003;
varSingle	=	\$0004;
varDouble	=	\$0005;
varCurrency	=	\$0006;
varDate	=	\$0007;
var0leStr	=	\$0008;
varDispatch	=	\$0009;
varError	=	\$000A;
varBoolean	=	\$000B;
varVariant	=	\$000C;
varUnknown	=	\$000D;
varByte	=	\$0011;
varString	=	\$0100;
varTypeMask	=	\$0FFF;
varArray	=	\$2000;
varByRef	=	\$4000;

Pascal Scripting: DirExists

Prototype:

function DirExists(const Name: String): Boolean;

Description:

Returns True if the specified directory name exists. The specified name may include a trailing backslash.

Pascal Scripting: FileExists

Prototype:
function FileExists(const Name: String): Boolean;

Description:

Returns True if the specified file exists.

Pascal Scripting: FileOrDirExists

Prototype:

function FileOrDirExists(const Name: String):
Boolean;

Description:

Returns True if the specified directory or file name exists. The specified name may include a trailing backslash.

Pascal Scripting: FileSize

Prototype:

function FileSize(const Name: String; var Size: Integer): Boolean;

Description:

Sets Size to the size of the specified file in bytes. Returns True if the file size was set successfully and False otherwise. Only supports file sizes smaller than 2 GB. To get all 64 bits of the file size, use <u>FindFirst/FindClose</u> instead.

Pascal Scripting: GetSpaceOnDisk

Prototype:

function GetSpaceOnDisk(const Path: String; const InMegabytes: Boolean; var Free, Total: Cardinal): Boolean;

Description:

Returns the number of free and total bytes or megabytes on a drive. Path specifies a directory on the drive or UNC share to check; it can be either the root (e.g. C:\) or an existing subdirectory. The setting of the InMegabytes parameter determines whether it returns figures in bytes or in megabytes (2^20), rounded down. Returns True if successful, False otherwise.

Remarks:

The figures returned by this function are capped at 2147483647 (2^31-1). Therefore, if InMegaBytes is False, it will return no more than 2147483647 bytes. If InMegaBytes is True, it will return no more than 2147483647 megabytes.

Example:

```
var
Path: String;
FreeMB, TotalMB: Cardinal;
begin
// Get and display free megabytes on the Program Fil
Path := ExpandConstant('{pf}');
if GetSpaceOnDisk(Path, True, FreeMB, TotalMB) then
begin
MsgBox('There are ' + IntToStr(FreeMB) + ' megabyt
Path, mbInformation, MB_OK);
end
else begin
// the function failed
end;
end;
```

See also:

GetSpaceOnDisk64

Pascal Scripting: GetSpaceOnDisk64

Prototype:

function GetSpaceOnDisk64(const Path: String; var Free, Total: Int64): Boolean;

Description:

Returns the number of free and total bytes on a drive. Path specifies a directory on the drive or UNC share to check; it can be either the root (e.g. $C: \$) or an existing subdirectory. Returns True if successful, False otherwise.

Remarks:

Requires Unicode Inno Setup.

Example:

```
var
  Path: String;
  FreeBytes, TotalBytes: Int64;
begin
  // Get and display free bytes on the Program Files (
  Path := ExpandConstant('{pf}');
  if GetSpaceOnDisk64(Path, FreeBytes, TotalBytes) the
  begin
    MsgBox('There are ' + IntToStr(FreeBytes) + ' byte
      Path, mbInformation, MB OK);
  end
  else begin
    // the function failed
  end;
end;
See also:
```

<u>GetSpaceOnDisk</u>

Pascal Scripting: FileSearch

Prototype:

function FileSearch(const Name, DirList: string):
String;

Description:

Searches through the directories passed in DirList for a file named Name. DirList should be directory names separated by semicolons. If FileSearch locates a file matching Name, it returns a string containing a fully-qualified path name for that file. If no matching file exists, FileSearch returns an empty string.

Pascal Scripting: FindFirst

Prototype:

function FindFirst(const FileName: String; var FindRec: TFindRec): Boolean;

Description:

Retrieves information about the first file matching the wildcard specified by FileName. Returns True if successful.

TFindRec is defined as:

```
TFindRec = record
  Name: String;
  Attributes: LongWord;
  SizeHigh: LongWord;
  SizeLow: LongWord;
  CreationTime: TFileTime; // time file was creat
  LastAccessTime: TFileTime; // time file was last
  LastWriteTime: TFileTime; // time file was last
AlternateName: String; // file's short name (
  FindHandle: THandle; // used internally
end;
```

TFileTime is defined as:

```
TFileTime = record
  dwLowDateTime: DWORD;
  dwHighDateTime: DWORD;
end;
```

Valid file attributes are:

FILE_ATTRIBUTE_READONLY FILE ATTRIBUTE HIDDEN FILE ATTRIBUTE SYSTEM FILE ATTRIBUTE DIRECTORY FILE ATTRIBUTE ARCHIVE FILE_ATTRIBUTE_DEVICE FILE_ATTRIBUTE_NORMAL FILE_ATTRIBUTE_TEMPORARY

```
// name of the found 1
       // file attributes
// size of the file, ι
      // size of the file, ]
```

```
FILE_ATTRIBUTE_SPARSE_FILE
FILE_ATTRIBUTE_REPARSE_POINT
FILE_ATTRIBUTE_COMPRESSED
FILE_ATTRIBUTE_OFFLINE
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED
FILE_ATTRIBUTE_ENCRYPTED
```

Remarks:

If FindFirst returns True, call <u>FindNext</u> to enumerate the rest of the files, and then <u>FindClose</u>.

Example:

The following example counts the number of files in the Windows System directory.

```
var
  FilesFound: Integer;
  FindRec: TFindRec;
begin
  FilesFound := 0;
  if FindFirst(ExpandConstant('{sys}\*'), FindRec) the
    try
      repeat
        // Don't count directories
        if FindRec.Attributes and FILE_ATTRIBUTE_DIRE(
          FilesFound := FilesFound + 1;
      until not FindNext(FindRec);
    finally
      FindClose(FindRec);
    end;
  end;
  MsgBox(IntToStr(FilesFound) + ' files found in the S
    mbInformation, MB OK);
end;
```

See also: FindNext FindClose

Pascal Scripting: FindNext

Prototype:

function FindNext(var FindRec: TFindRec): Boolean;

Description:

Retrieves information about the next matching file after a call to <u>FindFirst</u>. Returns True if successful.

Example:

For an example, see the documentation for <u>FindFirst</u>.

See also:

FindFirst FindClose

Pascal Scripting: FindClose

Prototype:

procedure FindClose(var FindRec: TFindRec);

Description:

Ends a find sequence, and frees the resources associated with it. You should always call this when <u>FindFirst</u> returns True.

Example:

For an example, see the documentation for <u>FindFirst</u>.

See also:

FindFirst FindNext

Pascal Scripting: GetCurrentDir

Prototype:
function GetCurrentDir: String;

Description:

Returns a string containing the name of the current directory.
Pascal Scripting: SetCurrentDir

Prototype:

function SetCurrentDir(const Dir: string): Boolean;

Description:

Sets the current directory. The return value is True if the current directory was successfully changed, or False if an error occurred.

Pascal Scripting: GetSystemDir

Prototype:

function GetSystemDir: String;

Description:

Returns fully qualified path of the Windows System directory. Only includes a trailing backslash if the Windows System directory is the root directory.

Pascal Scripting: GetSysWow64Dir

Prototype:

function GetSysWow64Dir: String;

Description:

64-bit Windows only: returns fully qualified path of the SysWOW64 directory. This is the actual directory in which 32-bit system files reside. An empty string is returned if this function is called on 32-bit Windows, or if for some reason it fails to retrieve the path on 64-bit Windows (unlikely).

Example:

var

```
S: String;
```

begin

```
S := GetSysWow64Dir;
```

// Must check the result -- an empty string is retur
// if there is no SysWOW64 directory.

```
if S <> '' then
```

MsgBox('SysWOW64 directory: ' + S, mbInformation, else

MsgBox('There is no SysWOW64 directory.', mbInform end;

Pascal Scripting: GetTempDir

Prototype:

function GetTempDir: String;

Description:

Returns fully qualified path of the temporary directory, with trailing backslash. This does not use the Win32 function GetTempPath, due to platform differences.

Gets the temporary file path as follows:

1. The path specified by the TMP environment variable.

2. The path specified by the TEMP environment variable, if TMP is not defined or if TMP specifies a directory that does not exist.

3. The Windows directory, if both TMP and TEMP are not defined or specify nonexistent directories.

Pascal Scripting: GetShellFolder

Prototype:

```
function GetShellFolder(Common: Boolean; const ID:
TShellFolderID): String;
```

Description:

Gets the location of the specified shell folder. Returns the 'common' version of the shell folder location if Common is True and the user has administrative privileges. On failure (unlikely but possible), an empty string is returned.

Remarks:

There is little reason to use this function. It is recommended that you use the ExpandConstant function instead to get the paths of shell folders.

Example:

```
var
Path: String;
begin
Path := GetShellFolder(False, sfAppData);
if Path <> '' then
begin
MsgBox('Application Data path = ' + Path, mbInform
end
else
begin
// handle failure
end;
end;
```

Pascal Scripting: GetShellFolderByCSIDL

Prototype:

function GetShellFolderByCSIDL(const Folder: Integer; const Create: Boolean): String;

Description:

Gets the path of the specified shell folder. Folder specifies the value of a CSIDL constant (a complete list of which can be found in ShIObj.h). If Create is True, the folder will be created if it does not exist. On failure, an empty string is returned.

Remarks:

It is recommended that you always specify True in the Create parameter. Otherwise, the function may fail if the CSIDL value is valid but the directory does not currently exist. (This is a Windows issue.)

Example:

```
const
  CSIDL_MYPICTURES = $0027;
. . .
var
  Path: String;
begin
  Path := GetShellFolderByCSIDL(CSIDL_MYPICTURES, True
  if Path <> '' then
  begin
    MsgBox('My Pictures path = ' + Path, mbInformatior
  end
  else
  begin
    // handle failure
  end;
end;
```

Pascal Scripting: GetShortName

Prototype:

function GetShortName(const LongName: String):
String;

Description:

Returns the short version of the specified long filename. If the short version of the long filename is not found, the long filename is returned.

Pascal Scripting: GenerateUniqueName

Prototype:

function GenerateUniqueName(Path: String; const Extension: String): String;

Description:

Generates a unique filename for a file in the specified path with the specified extension.

Pascal Scripting: GetVersionNumbers

Prototype:

function GetVersionNumbers(const Filename: String; var VersionMS, VersionLS: Cardinal): Boolean;

Description:

Gets the file version numbers of the specified file.

Pascal Scripting: GetVersionNumbersString

Prototype:

function GetVersionNumbersString(const Filename: String; var Version: String): Boolean;

Description:

Gets the file version numbers of the specified file, as a string.

Pascal Scripting: IsProtectedSystemFile

Prototype:

function IsProtectedSystemFile(const Filename: String): Boolean;

Description:

Returns True if the specified file is protected by Windows File Protection (and therefore can't be replaced).

Pascal Scripting: GetMD5OfFile

Prototype:

function GetMD50fFile(const Filename: String):
String;

Description:

Gets the MD5 sum of the specified file, as a string. An exception will be raised upon failure.

Pascal Scripting: GetSHA1OfFile

Prototype:

function GetSHA10fFile(const Filename: String):
String;

Description:

Gets the SHA-1 hash of the specified file, as a string. An exception will be raised upon failure.

Pascal Scripting: EnableFsRedirection

Prototype:

function EnableFsRedirection(const Enable: Boolean):
Boolean;

Description:

Controls whether built-in support functions that access files disable WOW64 file system redirection (with <u>some exceptions</u>). Specify True in the Enable parameter to leave redirection enabled when those functions are called; specify False to disable it. Returns the previous redirection state (True if redirection was enabled).

If False is passed in the Enable parameter and the user isn't running a supported 64-bit version of Windows, an exception will be raised. To avoid the exception, call <u>IsWin64</u> first.

Remarks:

After you've performed the operation that required changing the redirection state, be sure to restore the previous state. Always use a try..finally language construct to ensure that the previous state is restored even if an exception occurs. See below for an example.

By default, file system redirection is enabled in <u>32-bit mode</u> installs, and disabled in 64-bit mode installs.

This function has no effect on calls to functions in external DLLs. When invoking external functions, file system redirection is always left enabled.

It is legal to call this function with True in the Enable parameter if the user isn't running a 64-bit version of Windows. In such a case, the call has no effect.

Example:

The following example demonstrates how to launch an executable located in the 64-bit System directory. (Note: In a <u>64-bit mode</u> install, it isn't necessary to call EnableFsRedirection because file system redirection is already disabled by default.)

var

```
OldState: Boolean;
 ResultCode: Integer;
begin
 // First verify that the user is running a supported
 // of Windows, because calling EnableFsRedirection(F
 // raise an exception otherwise.
  if IsWin64 then
 begin
    // Turn off redirection, so that cmd.exe from the
    // directory is launched.
    OldState := EnableFsRedirection(False);
    try
      Exec(ExpandConstant('{cmd}'), '', '', SW_SHOW,
        ewWaitUntilTerminated, ResultCode);
    finally
      // Restore the previous redirection state.
      EnableFsRedirection(OldState);
    end;
 end;
end;
```

Pascal Scripting: Exec

Prototype:

function Exec(const Filename, Params, WorkingDir: String; const ShowCmd: Integer; const Wait: TExecWait; var ResultCode: Integer): Boolean;

Description:

Executes the specified executable or batch file, using the same credentials as Setup/Uninstall. The Wait parameter specifies whether the function should return immediately or wait until the launched process has terminated or is idle. Returns True if the specified file was executed successfully, False otherwise. If True is returned and Wait is ewWaitUntilTerminated then ResultCode returns the exit code of the process. If False is returned then ResultCode specifies the error that occurred. Use SysErrorMessage(ResultCode) to get a description of the error.

Remarks:

TExecWait is defined as:

```
TExecWait = (ewNoWait, ewWaitUntilTerminated,
ewWaitUntilIdle);
```

Use the <u>ShellExec</u> function instead if you need to launch a file that is not an executable or batch file.

Do not include quotes in the Filename parameter; the function will add them automatically.

The WorkingDir parameter can be an empty string, in which case it will try to extract a pathname from the Filename parameter and use that as the initial current directory for the process. If no pathname was specified in Filename, a default directory will be used.

If you have a single string containing both a filename and parameters (e.g. a command line obtained from an UninstallString registry value), you need not separate them yourself; just pass '>' in the Filename parameter, and the full command line in the Params parameter. (Note that when this is done, the function's special platform-independent support for .bat and .cmd files is disabled; it simply passes the specified

command line to CreateProcess without any processing.)

By default, when Setup/Uninstall is running in <u>64-bit mode</u>, this function disables WOW64 file system redirection when calling CreateProcess. It is possible to override this by calling <u>EnableFsRedirection</u>.

Example:

```
var
ResultCode: Integer;
begin
// Launch Notepad and wait for it to terminate
if Exec(ExpandConstant('{win}\notepad.exe'), '', '',
ewWaitUntilTerminated, ResultCode) then
begin
// handle success if necessary; ResultCode contair
end
else begin
// handle failure if necessary; ResultCode contair
end;
```

end;

See also:

ExecAsOriginalUser

Pascal Scripting: ExecAsOriginalUser

Prototype:

function ExecAsOriginalUser(const Filename, Params, WorkingDir: String; const ShowCmd: Integer; const Wait: TExecWait; var ResultCode: Integer): Boolean;

Description:

Executes the specified executable or batch file, using the (normally non-elevated) credentials of the user that started Setup initially. See <u>Exec</u> and the [Run] section flag runasoriginaluser for more information.

Remarks:

This function is not supported at uninstall time.

In very unusual failure cases (e.g. if the initial Setup process died unexpectedly), it is possible for this function to raise an exception instead of just returning False.

Pascal Scripting: ShellExec

Prototype:

function ShellExec(const Verb, Filename, Params, WorkingDir: String; const ShowCmd: Integer; const Wait: TExecWait; var ErrorCode: Integer): Boolean;

Description:

Opens the specified file or performs another action specified by Verb, using the same credentials as Setup/Uninstall. The filename can be an executable file, a document file, a folder, or a URL. Verb may be an empty string, in which case the default verb for the file type is used (usually "open"). The Wait parameter specifies whether the function should return immediately or wait until the launched process has terminated or is idle. Returns True if the specified file was opened successfully, False otherwise. If False is returned then ErrorCode specifies the error that occurred. Use SysErrorMessage(ErrorCode) to get a description of the error.

Remarks:

TExecWait is defined as:

```
TExecWait = (ewNoWait, ewWaitUntilTerminated,
ewWaitUntilIdle);
```

Note that passing a Wait value other than ewNoWait will have no effect if a new process isn't spawned (for example, if the file is opened inside an already-running instance of the program that handles the file type).

Refer to the system error codes on MSDN .

Example:

```
var
ErrorCode: Integer;
begin
if not ShellExec('', ExpandConstant('{app}\filename.
'', '', SW_SHOW, ewNoWait, ErrorCode) then
begin
   // handle failure if necessary
end;
```

end;

See also: ShellExecAsOriginalUser

Pascal Scripting: ShellExecAsOriginalUser

Prototype:

function ShellExecAsOriginalUser(const Verb, Filename, Params, WorkingDir: String; const ShowCmd: Integer; const Wait: TExecWait; var ErrorCode: Integer): Boolean;

Description:

Opens the specified file or performs another action specified by Verb, using the (normally non-elevated) credentials of the user that started Setup initially. See <u>ShellExec</u> and the [Run] section flag runasoriginaluser for more information.

Remarks:

This function is not supported at uninstall time.

In very unusual failure cases (e.g. if the initial Setup process died unexpectedly), it is possible for this function to raise an exception instead of just returning False.

Refer to the system error codes on MSDN .

Pascal Scripting: RenameFile

Prototype:

function RenameFile(const OldName, NewName: string):
Boolean;

Description:

Attempts to change the name of the file or directory specified by OldFile to NewFile. If the operation succeeds, RenameFile returns True. If it cannot rename the file (for example, if a file called NewName already exists), it returns False.

Pascal Scripting: FileCopy

Prototype:

function FileCopy(const ExistingFile, NewFile: String; const FailIfExists: Boolean): Boolean;

Description:

Copies ExistingFile to NewFile, preserving time stamp and file attributes.

If FailIfExists is True it will fail if NewFile already exists, otherwise it will overwrite it.

Returns True if successful, False otherwise.

Pascal Scripting: DeleteFile

Prototype:

function DeleteFile(const FileName: string):
Boolean;

Description:

Erases the file named by FileName from the disk. If the file cannot be deleted or does not exist, the function returns False.

Pascal Scripting: DelayDeleteFile

Prototype:

procedure DelayDeleteFile(const Filename: String; const Tries: Integer);

Description:

Attempts to delete Filename, retrying up to Tries times if the file is in use. It delays 250 msec between tries.

Pascal Scripting: SetNTFSCompression

Prototype:

function SetNTFSCompression(const FileOrDir: String; Compress: Boolean): Boolean;

Description:

Changes the NTFS compression state of a file or directory. Returns True if successful.

Remarks:

If a directory is specified, the compression state of any files present in the directory will not be changed.

Pascal Scripting: LoadStringFromFile

Prototype:

function LoadStringFromFile(const FileName: String; var S: AnsiString): Boolean;

Description:

Loads the specified binary or non Unicode text file into the specified string. Returns True if successful, False otherwise.

Pascal Scripting: LoadStringsFromFile

Prototype:

function LoadStringsFromFile(const FileName: String; var S: TArrayOfString): Boolean;

Description:

Loads the specified text file into the specified string array. Returns True if successful, False otherwise.

Pascal Scripting: SaveStringToFile

Prototype:

function SaveStringToFile(const FileName: String; const S: AnsiString; const Append: Boolean): Boolean;

Description:

Saves the specified string to the specified file. If Append is True and the specified file already exists, it will be appended to instead of overwritten. Returns True if successful, False otherwise.

Remarks:

This function does not automatically write a line break before or after the string. If Append is True and the existing file did not end in a line break, the function will effectively append to the existing last line. To avoid this you can put line break characters before and after your string:

```
SaveStringToFile('c:\filename.txt', #13#10 + 'the
string' + #13#10, True);
```

Pascal Scripting: SaveStringsToFile

Prototype:

function SaveStringsToFile(const FileName: String; const S: TArrayOfString; const Append: Boolean): Boolean;

Description:

Saves the specified string array to the specified file with ASCII encoding. If Append is True and the specified file already exists, it will be appended to instead of overwritten. Returns True if successful, False otherwise.

See also: SaveStringsToUTF8File

Pascal Scripting: SaveStringsToUTF8File

Prototype:

function SaveStringsToUTF8File(const FileName: String; const S: TArrayOfString; const Append: Boolean): Boolean;

Description:

Saves the specified string array to the specified file with UTF8 encoding. If Append is True and the specified file already exists, it will be appended to instead of overwritten. Returns True if successful, False otherwise.

Causes an internal error if called by non Unicode Setup or Uninstall.

See also: SaveStringsToFile

Pascal Scripting: CreateDir

Prototype:

function CreateDir(const Dir: string): Boolean;

Description:

Creates a new directory. The return value is True if a new directory was successfully created, or False if an error occurred.

Pascal Scripting: ForceDirectories

Prototype:

function ForceDirectories(Dir: string): Boolean;

Description:

Creates all the directories along the specified directory path all at once. If the first directories in the path do exist, but the latter ones don't, ForceDirectories creates just the ones that don't exist. Returns True if successful, False otherwise.

Pascal Scripting: RemoveDir

Prototype:

function RemoveDir(const Dir: string): Boolean;

Description:

Deletes an existing empty directory. The return value is True if a new directory was successfully deleted, or False if an error occurred.

Pascal Scripting: DelTree

Prototype:

function DelTree(const Path: String; const IsDir, DeleteFiles, DeleteSubdirsAlso: Boolean): Boolean;

Description:

Deletes the specified directory if IsDir is set to True, or files/directories matching a wildcard if IsDir is set to False. Returns True if it was able to successfully remove everything.

If DeleteFiles is set to True, files inside the specified directory will be deleted if IsDir is True, or files matching the specified wildcard (including those with hidden, system, and read-only attributes) will be deleted if IsDir is False.

If DeleteFiles and DeleteSubdirsAlso are both set to True, subdirectories (and their contents) will be deleted in addition to files.

Remarks:

This function will remove directories that are reparse points, but it will not recursively delete files/directories inside them.

Example:

```
begin
   // Delete the directory C:\Test and everything insic
   DelTree('C:\Test', True, True, True);
   // Delete files matching C:\Test\*.tmp
   DelTree('C:\Test\*.tmp', False, True, False);
   // Delete all files and directories inside C:\Test
```

// but leave the directory itself
DelTree('C:\Test*', False, True, True);
end;

Pascal Scripting: CreateShellLink

Prototype:

function CreateShellLink(const Filename, Description, ShortcutTo, Parameters, WorkingDir, IconFilename: String; const IconIndex, ShowCmd: Integer): String;

Description:

Creates a shortcut to a file or folder. Returns the resulting filename of the link, which may differ from Filename if it ended up creating a .pif file instead of a .lnk file. On failure, an exception will be raised.

Parameters: Filename

Filename of the shortcut file to be created. This should be the full path and must end with ".lnk".

Description

Description of the link. This will be displayed on Windows 2000/XP and other supporting OS when the user hovers the mouse over the file or shows the properties.

ShortcutTo

Target file for the shortcut. This must be the full path to the file. Double quotation marks to surround the path will be added automatically. *Parameters*

Parameters to pass to the target file of the shortcut. Parameters which may include spaces should have double quote marks surrounding them. e.g. ExpandConstant('"{app}\foo"')

WorkingDir

Working directory for the target file. This should be set to an absolute directory.

IconFilename

Path to file to supply the icon. If this is left as the empty string then the system default icon for the target file will be used.

IconIndex

Zero-based index of the icon.

ShowCmd

One of the SW_* constants
Remarks:

You will most likely want to remove this shortcut on uninstall. Do this by adding an entry to the UninstallDelete section.

```
CreateShellLink(
   ExpandConstant('{app}\config\Open licence database.]
   'Opens the licence database in SQLite',
   ExpandConstant('{app}\config\sqlite.exe'),
   ExpandConstant('"{app}\config\licences.db"'),
   ExpandConstant('{app}\config'),
   '',
   0,
   SW_SHOWNORMAL);
```

Pascal Scripting: UnpinShellLink

Prototype:

function UnpinShellLink(const Filename: String):
Boolean;

Description:

Attempt to unpin the shortcut with the specified filename. Returns True if the shortcut was successfully removed from the list of pinned items and/or the taskbar, or if the shortcut was not pinned at all.

Pascal Scripting: RegisterServer

Prototype:

procedure RegisterServer(const Is64Bit: Boolean; const Filename: String; const FailCriticalErrors: Boolean);

Description:

Registers the DLL/OCX with the specified filename. If Is64Bit is True, the DLL/OCX will be loaded as a 64-bit image and registered in a 64-bit process. If FailCriticalErrors is True, the system will not display any critical-error-handler message boxes. Raises an exception if not successful.

Example:

begin

// Register hhctrl.ocx located in the System directc RegisterServer(Is64BitInstallMode, ExpandConstant('{ end;

Because we specify Is64BitInstallMode in the first parameter, it will register the 64-bit OCX in the 64-bit System directory when Setup is running in 64-bit mode. Otherwise, it will register the 32-bit OCX in the 32-bit System directory.

Pascal Scripting: UnregisterServer

Prototype:

function UnregisterServer(const Is64Bit: Boolean; const Filename: String; const FailCriticalErrors: Boolean): Boolean;

Description:

Unregisters the DLL/OCX with the specified filename. If Is64Bit is True, the DLL/OCX will be loaded as a 64-bit image and unregistered in a 64-bit process. If FailCriticalErrors is True, the system will not display any critical-error-handler message boxes. Returns True if successful, False otherwise.

Pascal Scripting: RegisterTypeLibrary

Prototype:

procedure RegisterTypeLibrary(const Is64Bit: Boolean; const Filename: String);

Description:

Registers the type library with the specified filename. If Is64Bit is True, the type library will be registered in a 64-bit process. Raises an exception if not successful.

Example:

begin

// Register stdole2.tlb located in the System direct
 RegisterTypeLibrary(Is64BitInstallMode, ExpandConstate)
end;

Because we specify Is64BitInstallMode in the first parameter, it will register the 64-bit type library in the 64-bit System directory when Setup is running in 64-bit mode. Otherwise, it will register the 32-bit type library in the 32-bit System directory.

Pascal Scripting: UnregisterTypeLibrary

Prototype:

function UnregisterTypeLibrary(const Is64Bit: Boolean; const Filename: String): Boolean

Description:

Unregisters the type library with the specified filename. If Is64Bit is True, the type library will be unregistered in a 64-bit process. Returns True if successful, False otherwise.

Pascal Scripting: IncrementSharedCount

Prototype:

procedure IncrementSharedCount(const Is64Bit: Boolean; const Filename: String; const AlreadyExisted: Boolean);

Description:

Increments or initializes the reference count for the specified file in the following registry key:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersi

64-bit versions of Windows have two separate SharedDLLs keys, one for 64-bit files and one for 32-bit files. If Is64Bit is True, the 64-bit SharedDLLs key will be updated, otherwise the 32-bit SharedDLLs key will be updated. The setting of this parameter should correspond to the bitness of the file; for example, if it is a 32-bit DLL located in the 32-bit System directory, you should specify False. You may also specify Is64BitInstallMode in which case it will use the current install mode to determine which key to open.

Pass True in the AlreadyExisted parameter if the file already exists; in this case the initial reference count for the file will be 2 if the value for the file doesn't already exist in the registry. (This behavior is in line with Microsoft's requirements.)

An exception will be raised if the registry key cannot be opened for write access.

Pascal Scripting: DecrementSharedCount

Prototype:

function DecrementSharedCount(const Is64Bit: Boolean; const Filename: String): Boolean;

Description:

Decrements the reference count for the specified file in the following registry key:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersi

64-bit versions of Windows have two separate SharedDLLs keys, one for 64-bit files and one for 32-bit files. If Is64Bit is True, the 64-bit SharedDLLs key will be updated, otherwise the 32-bit SharedDLLs key will be updated. The setting of this parameter should correspond to the bitness of the file; for example, if it is a 32-bit DLL located in the 32-bit System directory, you should specify False. You may also specify Is64BitInstallMode in which case it will use the current install mode to determine which key to open.

Returns True if the count reached zero (meaning it's OK to delete the file). Returns False if the new count is greater than zero, or if the value for the file doesn't exist or is in an unrecognizable format.

An exception will be raised if the registry key cannot be opened for write access.

Pascal Scripting: RestartReplace

Prototype:

procedure RestartReplace(const TempFile, DestFile: String);

Description:

Renames TempFile to DestFile the next time Windows is started. If DestFile already existed, it will be overwritten. If DestFile is " then TempFile will be deleted. An exception will be raised upon failure.

Pascal Scripting: UnregisterFont

Prototype:

procedure UnregisterFont(const FontName, FontFilename: String);

Description:

Unregisters the font with the specified face and filename.

Pascal Scripting: ModifyPifFile

Prototype:

function ModifyPifFile(const Filename: String; const CloseOnExit: Boolean): Boolean;

Description:

Changes the "Close on exit" setting of a .pif file. Returns True if it was able to make the change.

Pascal Scripting: RegKeyExists

Prototype:

function RegKeyExists(const RootKey: Integer; const SubKeyName: String): Boolean;

Description:

Returns True if the specified registry key exists.

Example:

Pascal Scripting: RegValueExists

Prototype:

```
function RegValueExists(const RootKey: Integer;
const SubKeyName, ValueName: String): Boolean;
```

Description:

Returns True if the specified registry key and value exist.

Pascal Scripting: RegGetSubkeyNames

Prototype:

```
function RegGetSubkeyNames(const RootKey: Integer;
const SubKeyName: String; var Names:
TArrayOfString): Boolean;
```

Description:

Opens the specified registry key and reads the names of its subkeys into the specified string array Names. Returns True if successful, False otherwise.

```
var
  Names: TArrayOfString;
  I: Integer;
 S: String;
begin
  if RegGetSubkeyNames(HKEY_CURRENT_USER, 'Control Par
  begin
   S := '';
    for I := 0 to GetArrayLength(Names)-1 do
      S := S + Names[I] + #13#10;
    MsgBox('List of subkeys:'#13#10#13#10 + S, mbInfor
  end else
 begin
    // add any code to handle failure here
  end;
end;
```

Pascal Scripting: RegGetValueNames

Prototype:

```
function RegGetValueNames(const RootKey: Integer;
const SubKeyName: String; var Names:
TArrayOfString): Boolean;
```

Description:

Opens the specified registry key and reads the names of its values into the specified string array Names. Returns True if successful, False otherwise.

```
var
  Names: TArrayOfString;
  I: Integer;
 S: String;
begin
  if RegGetValueNames(HKEY_CURRENT_USER, 'Control Pane
  begin
   S := '';
    for I := 0 to GetArrayLength(Names)-1 do
      S := S + Names[I] + #13#10;
    MsgBox('List of values:'#13#10#13#10 + S, mbInforn
  end else
 begin
    // add any code to handle failure here
  end;
end;
```

Pascal Scripting: RegQueryStringValue

Prototype:

function RegQueryStringValue(const RootKey: Integer; const SubKeyName, ValueName: String; var ResultStr: String): Boolean;

Description:

Queries the specified REG_SZ- or REG_EXPAND_SZ-type value, and returns the data in ResultStr. Returns True if successful. When False is returned, ResultStr is unmodified.

Example:

var

```
Country: String;
```

begin

```
if RegQueryStringValue(HKEY_CURRENT_USER, 'Control F
    'sCountry', Country) then
```

begin

```
// Successfully read the value
```

```
MsgBox('Your country: ' + Country, mbInformation,
end;
```

end;

Pascal Scripting: RegQueryMultiStringValue

Prototype:

function RegQueryMultiStringValue(const RootKey: Integer; const SubKeyName, ValueName: String; var ResultStr: String): Boolean;

Description:

Queries the specified REG_MULTI_SZ-type registry value, and returns the data in ResultStr. Returns True if successful. When False is returned, ResultStr is unmodified.

Remarks:

In a REG_MULTI_SZ-type value, each string is separated by a null character (#0).

Pascal Scripting: RegQueryDWordValue

Prototype:

function RegQueryDWordValue(const RootKey: Integer; const SubKeyName, ValueName: String; var ResultDWord: Cardinal): Boolean;

Description:

Queries the specified REG_DWORD-type registry value, and returns the data in ResultDWord. Returns True if successful. When False is returned, ResultDWord is unmodified.

Example:

var

```
HistoryBufferSize: Cardinal;
```

begin

```
if RegQueryDWordValue(HKEY_CURRENT_USER, 'Console',
    'HistoryBufferSize', HistoryBufferSize) then
begin
   // Successfully read the value
   MsgBox('Console history buffer size: ' + IntToStr(
```

```
mbInformation, MB_OK);
```

end;

end;

Pascal Scripting: RegQueryBinaryValue

Prototype:

function RegQueryBinaryValue(const RootKey: Integer; const SubKeyName, ValueName: String; var ResultStr: AnsiString): Boolean;

Description:

Queries the specified REG_BINARY-type registry value, and returns the data in ResultStr. Returns True if successful. When False is returned, ResultStr is unmodified.

Pascal Scripting: RegWriteStringValue

Prototype:

function RegWriteStringValue(const RootKey: Integer; const SubKeyName, ValueName, Data: String): Boolean;

Description:

Writes the specified REG_SZ-type registry value. Returns True if successful, False otherwise.

Remarks:

If the value already exists and is of type REG_EXPAND_SZ, the new value will also be of type REG_EXPAND_SZ. Otherwise, a REG_SZ-type value will be created.

```
begin
```

```
RegWriteStringValue(HKEY_CURRENT_USER, 'Software\My
    'UserName', ExpandConstant('{sysuserinfoname}'));
end;
```

Pascal Scripting: RegWriteExpandStringValue

Prototype:

function RegWriteExpandStringValue(const RootKey: Integer; const SubKeyName, ValueName, Data: String): Boolean;

Description:

Writes the specified REG_EXPAND_SZ-type registry value. Returns True if successful, False otherwise.

Example:

begin

```
RegWriteStringValue(HKEY_CURRENT_USER, 'Software\My
   'UserName', '%UserName%);
```

end;

Pascal Scripting: RegWriteMultiStringValue

Prototype:

function RegWriteMultiStringValue(const RootKey: Integer; const SubKeyName, ValueName, Data: String): Boolean;

Description:

Writes the specified REG_MULTI_SZ-type registry value. Returns True if successful, False otherwise.

Remarks:

In a REG_MULTI_SZ-type value, each string is separated by a null character (#0).

Example:

begin

RegWriteMultiStringValue(HKEY_CURRENT_USER, 'Softwar 'MultiStringTest', 'String1' + #0 + 'String2' + #0 end;

Pascal Scripting: RegWriteDWordValue

Prototype:

function RegWriteDWordValue(const RootKey: Integer; const SubKeyName, ValueName: String; const Data: Cardinal): Boolean;

Description:

Writes the specified REG_DWORD-type registry value. Returns True if successful, False otherwise.

Example:

begin

end;

Pascal Scripting: RegWriteBinaryValue

Prototype:

function RegWriteBinaryValue(const RootKey: Integer; const SubKeyName, ValueName: String; const Data: AnsiString): Boolean;

Description:

Writes the specified REG_BINARY-type registry value. Returns True if successful, False otherwise.

Example:

begin

```
RegWriteBinaryValue(HKEY_CURRENT_USER, 'Software\My
            'BinaryTest', 'Whatever' + #1#2#3#4);
end;
```

Pascal Scripting: RegDeleteKeyIncludingSubkeys

Prototype:

function RegDeleteKeyIncludingSubkeys(const RootKey: Integer; const SubkeyName: String): Boolean;

Description:

Deletes the specified key and all subkeys. Returns True if successful, False otherwise.

Pascal Scripting: RegDeleteKeyIfEmpty

Prototype:

function RegDeleteKeyIfEmpty(const RootKey: Integer; const SubkeyName: String): Boolean;

Description:

Deletes the specified subkey if it has no subkeys or values. Returns True if successful, False otherwise.

Pascal Scripting: RegDeleteValue

Prototype:

function RegDeleteValue(const RootKey: Integer; const SubKeyName, ValueName: String): Boolean;

Description:

Deletes the specified value. Returns True if successful, False otherwise.

Pascal Scripting: IniKeyExists

Prototype:
function IniKeyExists(const Section, Key, Filename: String): Boolean;

Description:

Returns True if the specified INI key exists.

Pascal Scripting: IsIniSectionEmpty

Prototype:

function IsIniSectionEmpty(const Section, Filename: String): Boolean;

Description:

Returns True if the specified INI section is empty.

Pascal Scripting: GetIniBool

Prototype:

```
function GetIniBool(const Section, Key: String;
const Default: Boolean; const Filename: String):
Boolean
```

Description:

Reads a Boolean from an INI file.

Pascal Scripting: GetIniInt

Prototype:

function GetIniInt(const Section, Key: String; const Default, Min, Max: Longint; const Filename: String): Longint;

Description:

Reads a Longint from an INI file. If the Longint read is not between Min/Max then it returns Default. If Min=Max then Min/Max are ignored.

Pascal Scripting: GetIniString

Prototype:

function GetIniString(const Section, Key, Default, Filename: String): String;

Description:

Reads a String from an INI file.

Pascal Scripting: SetIniBool

Prototype:

function SetIniBool(const Section, Key: String; const Value: Boolean; const Filename: String): Boolean;

Description:

Writes a Boolean to an INI file.

Pascal Scripting: SetIniInt

Prototype:

function SetIniInt(const Section, Key: String; const Value: Longint; const Filename: String): Boolean;

Description:

Writes a Longint to an INI file.

Pascal Scripting: SetIniString

Prototype:

function SetIniString(const Section, Key, Value, Filename: String): Boolean;

Description:

Writes a string to an INI file.

Pascal Scripting: DeleteIniSection

Prototype:

procedure DeleteIniSection(const Section, Filename: String);

Description:

Deletes the specified section from an INI file.
Pascal Scripting: DeleteIniEntry

Prototype:

procedure DeleteIniEntry(const Section, Key, Filename: String);

Description:

Deletes the specified key from an INI file.

Pascal Scripting: CreateInputQueryPage

Prototype:

function CreateInputQueryPage(const AfterID: Integer; const ACaption, ADescription, ASubCaption: String): TInputQueryWizardPage;

Description:

Creates a wizard page containing edit boxes.

Remarks:

To create edit boxes on the page, call the Add method. Use the Values property to get/set the text of the edit boxes.

Example:

```
var
  Page: TInputQueryWizardPage;
  UserName, UserCompany: String;
. . .
// Create the page
Page := CreateInputQueryPage(wpWelcome,
  'Personal Information', 'Who are you?',
  'Please specify your name and the company for whom y
// Add items (False means it's not a password edit)
Page.Add('Name:', False);
Page.Add('Company:', False);
// Set initial values (optional)
Page.Values[0] := ExpandConstant('{sysuserinfoname}');
Page.Values[1] := ExpandConstant('{sysuserinfoorg}');
. . .
// Read values into variables
UserName := Page.Values[0];
UserCompany := Page.Values[1];
```

See also:

TInputQueryWizardPage

Pascal Scripting: CreateInputOptionPage

Prototype:

function CreateInputOptionPage(const AfterID: Integer; const ACaption, ADescription, ASubCaption: String; Exclusive, ListBox: Boolean): TInputOptionWizardPage;

Description:

Creates a wizard page containing check boxes or radio buttons.

If Exclusive is True, radio buttons are displayed instead of check boxes, and only one item in the list may be selected at a time. If ListBox is True, the check boxes or radio buttons are placed inside a scrollable list box.

Remarks:

To create check boxes / radio buttons on the page, call the Add method. Use the Values property to get/set the checked state of items. On pages created with Exclusive=True, you can get/set the index of the one selected item via the SelectedValueIndex property.

Example:

```
var
Page: TInputOptionWizardPage;
IsRegisteredUser: Boolean;
...
// Create the page
Page := CreateInputOptionPage(wpWelcome,
   'License Information', 'Are you a registered user?',
   'If you are a registered user, please check the box
False, False);
// Add items
Page.Add('I am a registered user');
// Set initial values (optional)
```

Page.Values[0] := False;

. . .

// Read values into variables
IsRegisteredUser := Page.Values[0];

See also:

TInputOptionWizardPage

Pascal Scripting: CreateInputDirPage

Prototype:

function CreateInputDirPage(const AfterID: Integer; const ACaption, ADescription, ASubCaption: String; AAppendDir: Boolean; ANewFolderName: String): TInputDirWizardPage;

Description:

Creates a wizard page that contains edit boxes and Browse buttons for selecting directories. If AAppendDir is True, the value of ANewFolderName will be appended onto any folder name the user clicks. If AAppendDir is False and ANewFolderName is not empty, a Make New Folder button will be shown that creates a new folder with the specified default name.

Remarks:

To create directory selection boxes on the page, call the Add method. Use the Values property to get/set the items' values.

Example:

```
var
Page: TInputDirWizardPage;
DataDir: String;
...
// Create the page
Page := CreateInputDirPage(wpWelcome,
'Select Personal Data Location', 'Where should personal data files will be stored in the following
'To continue, click Next. If you would like to select
False, 'New Folder');
// Add item (with an empty caption)
Page.Add('');
// Set initial value (optional)
Page.Values[0] := ExpandConstant('{userappdata}\My Compare the following the fo
```

. . .

// Read value into variable
DataDir := Page.Values[0];

See also:

<u>TInputDirWizardPage</u>

Pascal Scripting: CreateInputFilePage

Prototype:

function CreateInputFilePage(const AfterID: Integer; const ACaption, ADescription, ASubCaption: String): TInputFileWizardPage;

Description:

Creates a wizard page that contains edit boxes and Browse buttons for selecting files.

Remarks:

To create file selection boxes on the page, call the Add method. Use the Values property to get/set the items' values.

An example Filter: 'Text files (*.txt)|*.txt|All files (*.*)|*.*'

Example:

```
var
Page: TInputFileWizardPage;
NotepadLocation: String;
```

```
. . .
```

```
// Create the page
Page := CreateInputFilePage(wpWelcome,
   'Select Notepad Location', 'Where is Notepad located
   'Select where Notepad is located, then click Next.')
// Add item
Page.Add('Location of notepad.exe:', // captic
   'Executable files|*.exe|All files|*.*', // filter
   '.exe'); // defaul
// Set initial value (optional)
Page.Values[0] := ExpandConstant('{win}\notepad.exe');
```

. . .

// Read value into variable
NotepadLocation := Page.Values[0];

See also: <u>TInputFileWizardPage</u>

Pascal Scripting: CreateOutputMsgPage

Prototype:

function CreateOutputMsgPage(const AfterID: Integer; const ACaption, ADescription, AMsg: String): TOutputMsgWizardPage;

Description:

Creates a wizard page containing only static text. The AMsg parameter specifies the text to display.

Example:

var

Page: TOutputMsgWizardPage;

. . .

// Create the page

Page := CreateOutputMsgPage(wpWelcome,

```
'Information', 'Please read the following important 'Blah blah blah.');
```

See also:

<u>TOutputMsgWizardPage</u>

Pascal Scripting: CreateOutputMsgMemoPage

Prototype:

function CreateOutputMsgMemoPage(const AfterID: Integer; const ACaption, ADescription, ASubCaption: String; const AMsg: AnsiString): TOutputMsgMemoWizardPage;

Description:

Creates a wizard page containing static text as well as a read-only, multi-line edit control, capable of displaying RTF text. The ASubCaption parameter specifies the static text to display. AMsg specifies the text to assign to the edit control.

Example:

var

Page: TOutputMsgMemoWizardPage;

. . .

```
// Create the page
```

```
Page := CreateOutputMsgMemoPage(wpWelcome,
```

```
'Information', 'Please read the following important
'When you are ready to continue with Setup, click Ne
'Blah blah blah.');
```

See also:

<u>TOutputMsgMemoWizardPage</u>

Pascal Scripting: CreateOutputProgressPage

Prototype:

function CreateOutputProgressPage(const ACaption, ADescription: String): TOutputProgressWizardPage;

Description:

Creates a wizard page containing static text as well as a progress bar (which is hidden by default).

Unlike the other types of wizard pages, progress pages are not displayed as part of the normal page sequence (note that there is no AfterID parameter). A progress page can only be displayed programmatically by calling its Show method.

Remarks:

Call the Show method to activate and show the page. When you're finished with it, call the Hide method to revert to the previous page.

Always put the Hide call inside the finally part of a try..finally language construct, as demonstrated in *CodeDlg.iss*. Not calling Hide will result in the wizard being permanently stuck on the progress page.

To set the text on the page, call the SetText method. SetText takes two string parameters: use the first to tell the user what you're doing, and the second to display a file or directory name. Either parameter may be blank.

To display or update the progress bar, call the SetProgress method. SetProgress takes two integer parameters: the first specifies the position of the progress bar (zero-based), and the second specifies the highest possible position. If the second parameter is 0, the progress bar will be hidden.

Example:

See *CodeDlg.iss* for an example.

See also:

<u>TOutputProgressWizardPage</u>

Pascal Scripting: CreateCustomPage

Prototype:

function CreateCustomPage(const AfterID: Integer; const ACaption, ADescription: String): TWizardPage;

Description:

Creates a custom wizard page. The page is empty by default; you have to create your own controls afterward and place them on the page (by setting their Parent properties to the Surface property of the <u>TWizardPage</u> instance returned by this function).

Example:

See CodeClasses.iss for an example.

See also: TWizardPage

Pascal Scripting: CreateCustomForm

Prototype:

function CreateCustomForm: TSetupForm;

Description:

Creates a form. The form is empty by default; you have to create your own controls afterward and place them on the form (by setting their Parent properties to the <u>TSetupForm</u> instance returned by this function).

Remarks:

You should call this function instead of creating <u>TForm</u> or <u>TSetupForm</u> instances directly. This function automatically initializes the font and other properties of the created form to be like Setup's other dialogs.

The [LangOptions] section's DialogFontName and DialogFontSize directives determine the font used by the form and, by default, any child controls created on the form.

Example:

See CodeClasses.iss for an example.

See also:

TForm TSetupForm

Pascal Scripting: PageFromID

Prototype:

function PageFromID(const ID: Integer): TWizardPage;

Description:

Given a page ID, returns a <u>TWizardPage</u> instance. Call this if, for example, you need to get at the surface of a page and only know its ID.

An exception will be raised if an invalid page ID is specified.

Example:

```
var
Page: TWizardPage;
begin
Page := PageFromID(wpWelcome);
Page.Surface.Color := clBlue;
end;
```

See also: TWizardPage

Pascal Scripting: PageIndexFromID

Prototype:

function PageIndexFromID(const ID: Integer):
Integer;

Description:

Given a page ID, returns an position index. Call this if, for example, you want to check whether a page is positioned before a certain other page.

Example:

```
function ShouldSkipPage(PageID: Integer): Boolean;
begin
```

```
Result := PageIndexFromID(PageID) < PageIndexFromID(
end;</pre>
```

Pascal Scripting: ScaleX

Prototype:

function ScaleX(X: Integer): Integer;

Description:

Takes an X coordinate or width and returns it scaled to fit the size of the current dialog font. If the dialog font is 8-point MS Sans Serif and the user is running Windows in Small Fonts (96 dpi), then X is returned unchanged.

Pascal Scripting: ScaleY

Prototype:

function ScaleY(Y: Integer): Integer;

Description:

Takes a Y coordinate or height and returns it scaled to fit the size of the current dialog font. If the dialog font is 8-point MS Sans Serif and the user is running Windows in Small Fonts (96 dpi), then Y is returned unchanged.

Pascal Scripting: MsgBox

Prototype:

function MsgBox(const Text: String; const Typ: TMsgBoxType; const Buttons: Integer): Integer;

Description:

Displays a message box. Text specifies the message to display. Typ specifies which icon to use in the message box. Buttons specifies which buttons to include in the message box. Returns an ID* constant indicating the button the user clicked, or 0 if the function fails (which shouldn't happen unless an invalid parameter is specified or system resources are exhausted).

Remarks:

TMsgBoxType is defined as:

```
TMsgBoxType = (mbInformation, mbConfirmation,
mbError, mbCriticalError);
```

Example:

```
begin
  // Display a simple message box with an OK button
  MsgBox('Hello.', mbInformation, MB_OK);
  // Ask the user a Yes/No question
  if MsgBox('Are you sure?', mbConfirmation, MB_YESNO)
  begin
    // user clicked Yes
  end;
  // Ask the user a Yes/No question, defaulting to No
  if MsgBox('Are you sure?', mbConfirmation, MB_YESNO
  begin
    // user clicked Yes
  end;
end;
```

Pascal Scripting: SuppressibleMsgBox

Prototype:

function SuppressibleMsgBox(const Text: String; const Typ: TMsgBoxType; const Buttons, Default: Integer): Integer;

Description:

Displays a suppressible message box. If message boxes are being suppressed (see <u>Setup Command Line Parameters</u>), Default is returned. Otherwise, SuppressibleMsgBox acts the same as the regular <u>MsgBox</u>.

Pascal Scripting: GetOpenFileName

Prototype:

function GetOpenFileName(const Prompt: String; var FileName: String; const InitialDirectory, Filter, DefaultExtension: String): Boolean;

Description:

Displays a dialog box that enables the user to select an existing file. Returns True if the user selected a file, False otherwise. The name of the selected file is returned in the FileName string.

Remarks:

```
An example Filter: 'Text files (*.txt)|*.txt|All files (*.*)|*.*'
```

Example:

```
var
Filename: String;
begin
// Set the initial filename
Filename := '';
if GetOpenFileName('', Filename, '',
    'Text Documents (*.txt)|*.txt|All Files|*.*', 't>
    begin
    // Successful; user clicked OK
    // Filename contains the selected filename
end;
end;
```

Pascal Scripting: GetSaveFileName

Prototype:

function GetSaveFileName(const Prompt: String; var FileName: String; const InitialDirectory, Filter, DefaultExtension: String): Boolean;

Description:

Displays a dialog box that enables the user to select a new file. Returns True if the user selected a file, False otherwise. The name of the selected file is returned in the FileName string.

Remarks:

```
An example Filter: 'Text files (*.txt)|*.txt|All files (*.*)|*.*'
```

Example:

```
var
Filename: String;
begin
// Set the initial filename
Filename := '';
if GetSaveFileName('', Filename, '',
    'Text Documents (*.txt)|*.txt|All Files|*.*', 't>
    begin
    // Successful; user clicked OK
    // Filename contains the selected filename
end;
end;
```

Pascal Scripting: BrowseForFolder

Prototype:

function BrowseForFolder(const Prompt: String; var Directory: String; const NewFolderButton: Boolean): Boolean;

Description:

Displays a dialog box that enables the user to select a directory. The current value of Directory is used as the initially selected directory. If NewFolderButton is True, a *New Folder* button will be shown, allowing the user to create new folders. Returns True if the user selected a directory and clicked OK, False otherwise. The selected directory is returned in the Directory string.

Remarks:

On Windows versions prior to XP, passing False in the NewFolderButton parameter has no effect; the New Folder button will always be shown. This is a Windows limitation.

Pascal Scripting: ExitSetupMsgBox

Prototype:

function ExitSetupMsgBox: Boolean;

Description:

Displays the "Exit Setup?" message box, and returns True if the user selects Yes. Does not terminate Setup or Uninstall.

Pascal Scripting: CreateOleObject

Prototype:

function CreateOleObject(const ClassName: string):
Variant;

Description:

See the <u>Using COM Automation objects</u> topic.

Pascal Scripting: GetActiveOleObject

Prototype:

function GetActiveOleObject(const ClassName: string): Variant;

Description:

See the <u>Using COM Automation objects</u> topic.

Pascal Scripting: IDispatchInvoke

Prototype:

```
function IDispatchInvoke(Self: IDispatch;
PropertySet: Boolean; const Name: String; Par: array
of Variant): Variant;
```

Description:

Use IDispatchInvoke to access a COM Automation property or method whose name is a reserved word.

Example:

var

AObject: Variant;

AType: String;

begin

```
AObject := CreateOleObject('MyObject');
```

```
// Set a property named 'Type'
```

```
// Cannot use "AObject.Type := 'MyType';" because Ty
IDispatchInvoke(AObject, True, 'Type', ['MyType']);
// Get a property or call a method named 'Type'
AType := IDispatchInvoke(AObject, False, 'Type', [''
end;
```

See also:

Using COM Automation objects

Pascal Scripting: CreateComObject

Prototype:

function CreateComObject(const ClassID: TGUID): IUnknown;

Description:

See the <u>Using COM Automation objects</u> topic.

Pascal Scripting: StringToGUID

Prototype:

function StringToGUID(const S: String): TGUID;

Description:

StringToGUID converts the string representation of a GUID into a 'real' GUID. An exception will be raised upon failure.

See also:

Using COM Automation objects

Pascal Scripting: OleCheck

Prototype:

procedure OleCheck(Result: HResult);

Description:

Use OleCheck to wrap any IUnknown based COM methods you call, so that if that method fails, an exception will be raised.

See also:

Using COM Automation objects

Pascal Scripting: CoFreeUnusedLibraries

Prototype:

procedure CoFreeUnusedLibraries;

Description:

See the Using COM Automation objects topic.

Pascal Scripting: Log

Prototype:

procedure Log(const S: String);

Description:

Logs the specified string in Setup's log file.

Remarks:

Calls to this function are ignored if logging is not enabled via the <u>/LOG</u> command line parameter or the <u>SetupLogging</u> [Setup] section directive.

Pascal Scripting: Sleep

Prototype:

procedure Sleep(const Milliseconds: LongInt);

Description:

Suspends the execution of Setup or Uninstall for a specified interval.

Pascal Scripting: Random

Prototype:
function Random(const Range: Integer): Integer;

Description:

Returns a random number within the range $0 \le X \le Range$.

Pascal Scripting: Beep

Prototype:
procedure Beep;

Description:

Beeps.

Example: Beep; //Beeps

Pascal Scripting: BringToFrontAndRestore

Prototype:

procedure BringToFrontAndRestore;

Description:

Makes sure that Setup or Uninstall is visible and the foreground window.

Pascal Scripting: LoadDLL

Prototype:

function LoadDLL(const DLLName: String; var ErrorCode: Integer): Longint;

Description:

Loads the specified DLL. Returns the DLL handle if the DLL was loaded successfully, zero otherwise. If zero is returned then ErrorCode specifies the error that occurred. Use SysErrorMessage(ErrorCode) to get a description of the error.

Remarks:

This function is deprecated. See the <u>Using DLLs</u> topic.

Refer to the system error codes on MSDN .

Pascal Scripting: CallDLLProc

Prototype:

function CallDLLProc(const DLLHandle: Longint; const ProcName: String; const Param1, Param2: Longint; var Result: Longint): Boolean;

Description:

Calls the specified function in a DLL specified using the DLL handle returned by LoadDLL. Returns True is the procedure was called successfully, False otherwise.

The function must use the standard calling convention, accept two 4 byte integer parameters and return a 4 byte integer result.

Remarks:

Pascal Scripting: FreeDLL

Prototype:

function FreeDLL(const DLLHandle: Longint): Boolean;

Description:

Unloads a DLL specified using the DLL handle returned by LoadDLL.

Remarks:

Pascal Scripting: CastStringToInteger

Prototype:

function CastStringToInteger(var S: String):
Longint;

Description:

Casts a string to an integer so that a string can be passed to a DLL using CallDllProc.

Remarks:

Pascal Scripting: CastIntegerToString

Prototype:

function CastIntegerToString(const L: Longint):
String;

Description:

Casts an integer to a string so that a string can be received from a DLL using CallDIIProc.

Remarks: