

## What is Inno Setup?

**Inno Setup version 5.0.8**

**Copyright (C) 1997-2005 Jordan Russell. All rights reserved.**

**Portions Copyright (C) 2000-2005 Martijn Laan. All rights reserved.**

[Contacting Me](#)

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 all 32-bit Windows versions in use today -- Windows 95, 98, 2000, 2003, XP, Me, and NT 4.0. (No service packs are required.)
- Supports creation of a single EXE to install your program for easy online distribution. Disk spanning is also supported.
- Standard Windows 2000/XP-style wizard interface.
- Customizable setup types, e.g. Full, Minimal, Custom.
- Complete uninstall capabilities.
- Installation of files:  
Includes integrated support for "deflate", bzip2, and 7-Zip LZMA file compression. 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 registry and .INI entries.
- Integrated Pascal scripting engine.
- Support for multilingual installs.
- Support for passworded and encrypted installs.
- Silent install and silent uninstall.
- Full source code is available (Borland Delphi 2.0-5.0).

### ***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

- "Windows 98/NT 4+" This is shorthand for "Windows 98, 2000, XP, NT 4.0, Me, and later."
- "Windows NT" Whenever Windows NT is mentioned, it includes Windows 2000 and XP (which are NT 5), unless otherwise indicated.
- `monospaced text` When you see monospaced text in the documentation, it refers to text you would type in a script 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!)

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 Samples 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 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 myself here...
```

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.

### See also

[Parameters in Sections](#)

[Constants](#)

[\[Setup\] 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], and [LangOptions], contain lines separated into *parameters*. The following is an example of a [Files] section:

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

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 Windows System directory (System32 on Windows NT platforms).

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".

### {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:". On Windows NT platforms, this directory constant is equivalent to the *SystemDrive* environment variable.

### {pf}

Program Files. The path of the system's Program Files directory, typically "C:\Program Files".

### {cf}

Common Files. The path of the system's Common Files directory, typically "C:\Program Files\Common Files".

### {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\IS-xxxxx.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`.

## 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.

The "user" constants below refer to the currently logged in user's profile. "common" constants refer to the *All Users* profile.

Except where otherwise noted, shell folder constants work on all versions of Windows that Inno Setup supports, including Windows 95 and NT 4.0.

\* = The "common" form of this constant is mapped to the "user" form if the logged-in user lacks administrative privileges, or if the operating system is Windows 95/98/Me.

**{group}**

The path to the Start Menu folder, as selected by the user on Setup's *Select Start Menu Folder* wizard page. On Windows NT/2000/XP/2003, this folder is always created under the *All Users* profile unless the user installing the application does not have administrative privileges, in which case it is created on 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.

**{userdesktop} & {commondesktop} \***

The path to the desktop folder.

**{userdocs} & {commondocs}**

The path to the My Documents folder (or on NT 4.0, the Personal folder).

**{userfavorites} & {commonfavorites} \***

The path to the Favorites folder. Usage of these constants requires a `MinVersion` setting of at least "4.1, 4". Only Windows 2000 and later supports `{commonfavorites}`; if used on previous Windows versions, it will translate to the same directory as `{userfavorites}`.

**{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. Only Windows 2000 and later supports `{commontemplates}`; if used on previous Windows versions, it will translate to the same directory as `{usertemplates}`.

## 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. On Windows NT/2000/XP/2003, this is `Windows\System32\cmd.exe`. On Windows 95/98/Me, this is `Windows\COMMAND.COM`. 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 `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 wizard 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 Program}
```

**{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 Program}
```

#### **{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).

## Common Parameters

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

### Languages

*Description:*

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 installed, unless other parameters say it shouldn't.

*Example:*

```
Languages: en nl
```

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

### MinVersion

*Description:*

A minimum Windows version and Windows NT version respectively for the entry to be processed. If you use "0" for one of the versions then the entry will never be processed on that platform. Build numbers and/or service pack levels may be included in the version numbers. This overrides any `MinVersion` directive in the script's `[Setup]` section.

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

*Example:*

```
MinVersion: 4.0,4.0
```

### OnlyBelowVersion

*Description:*

Basically the opposite of `MinVersion`. Specifies the minimum Windows and Windows NT version for the entry *not* to be processed. For example, if you put 4.1, 5.0 and the user is running Windows 95 or NT 4.0 the entry *will* be processed, but if the user is running Windows 98 (which reports its version as 4.1) or Windows 2000 (which reports its version as NT 5.0), it will *not* be processed. Putting "0" for one of the versions means there is no upper version limit. Build numbers and/or service pack levels may be included in the version numbers. This overrides any `OnlyBelowVersion` directive in the script's `[Setup]` section.

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

*Example:*

```
OnlyBelowVersion: 4.1,5.0
```

## Components and Tasks Parameters

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

### Components

*Description:*

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 Components parameter is always installed, unless other parameters say it shouldn't.

*Example:*

```
[Files]
Source: "MYPROG.EXE"; DestDir: "{app}"; Components: main
Source: "MYPROG.HLP"; DestDir: "{app}"; Components: help
Source: "README.TXT"; DestDir: "{app}"
```

### Tasks

*Description:*

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 installed, unless other parameters say it shouldn't.

The *Don't create any icons* checkbox doesn't control [Icons] entries that have a Task parameter since these have their own checkboxes. Therefore Setup will change the *Don't create any icons* text to *Don't create a Start Menu folder* if you have any icons with a Task parameter.

*Example:*

```
[Icons]
Name: "{group}\My Program"; Filename: "{app}\MyProg.exe"; Components: main;
Tasks: startmenu
Name: "{group}\My Program Help"; Filename: "{app}\MyProg.hlp"; Components:
help; Tasks: startmenu
Name: "{userdesktop}\My Program"; Filename: "{app}\MyProg.exe"; Components:
main; Tasks: desktopicon
```

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 b
Name: r; Description: not a or b; Components: not a or b
Name: s; Description: not (a or b); Components: not (a or b)
Name: t; Description: a or b - old style; Components: a b
```

## [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
AppVerName=My Program version 1.4
DefaultDirName={pf}\My Program
DefaultGroupName=My Program
```

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

(**bold** = required)

### Compiler-related

- Compression
- DiskClusterSize
- DiskSliceSize
- DiskSpanning
- Encryption
- InternalCompressLevel
- MergeDuplicateFiles
- OutputBaseFilename
- OutputDir
- OutputManifestFile
- ReserveBytes
- SlicesPerDisk
- SolidCompression
- SourceDir
- UseSetupLdr
- VersionInfoCompany
- VersionInfoDescription
- VersionInfoTextVersion
- VersionInfoVersion

### Installer-related

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

- AllowCancelDuringInstall
- AllowNoIcons
- AllowRootDirectory
- AllowUNCPath
- AlwaysRestart
- AlwaysShowComponentsList
- AlwaysShowDirOnReadyPage
- AlwaysShowGroupOnReadyPage
- AlwaysUsePersonalGroup
- AppendDefaultDirName
- AppendDefaultGroupName
- AppComments
- AppContact
- AppId

- AppModifyPath
- AppMutex
- **AppName**
- AppPublisher
- AppPublisherURL
- AppReadmeFile
- AppSupportURL
- AppUpdatesURL
- AppVersion
- **AppVerName**
- ChangesAssociations
- ChangesEnvironment
- CreateAppDir
- CreateUninstallRegKey
- **DefaultDirName**
- DefaultGroupName
- DefaultUserInfoName
- DefaultUserInfoOrg
- DefaultUserInfoSerial
- DirExistsWarning
- DisableDirPage
- DisableFinishedPage
- DisableProgramGroupPage
- DisableReadyMemo
- DisableReadyPage
- DisableStartupPrompt
- EnableDirDoesntExistWarning
- ExtraDiskSpaceRequired
- InfoAfterFile
- InfoBeforeFile
- LanguageDetectionMethod
- LicenseFile
- MinVersion
- OnlyBelowVersion
- Password
- PrivilegesRequired
- RestartIfNeededByRun
- ShowLanguageDialog
- TimeStampRounding
- TimeStampsInUTC
- TouchDate
- TouchTime
- Uninstallable
- UninstallDisplayIcon
- UninstallDisplayName
- UninstallFilesDir
- UninstallLogMode
- UninstallRestartComputer
- UpdateUninstallLogAppName
- UsePreviousAppDir
- UsePreviousGroup
- UsePreviousSetupType
- UsePreviousTasks
- UsePreviousUserInfo
- UserInfoPage

**Cosmetic:** These directives are used only for display purposes in the Setup program.

- [AppCopyright](#)
- [BackColor](#)
- [BackColor2](#)
- [BackColorDirection](#)
- [BackSolid](#)
- [FlatComponentsList](#)
- [SetupIconFile](#)
- [ShowComponentSizes](#)
- [ShowTasksTreeLines](#)
- [WindowShowCaption](#)
- [WindowStartMaximized](#)
- [WindowResizable](#)
- [WindowVisible](#)
- [WizardImageBackColor](#)
- [WizardImageFile](#)
- [WizardImageStretch](#)
- [WizardSmallImageFile](#)

## **Obsolete**

- [AdminPrivilegesRequired](#)
- [AlwaysCreateUninstallIcon](#)
- [DisableAppendDir](#)
- [DontMergeDuplicateFiles](#)
- [MessagesFile](#)
- [UninstallIconFile](#)
- [UninstallIconName](#)
- [UninstallStyle](#)
- [WizardSmallImageBackColor](#)
- [WizardStyle](#)

## [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 installation"
Name: "custom"; Description: "Custom installation"; Flags: iscustom
```

The following is a list of the supported parameters:

### **Name** (*Required*)

#### *Description:*

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*)

#### *Description:*

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

#### *Example:*

```
Description: "Full installation"
```

### **Flags**

#### *Description:*

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.

#### *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 [Components and Tasks Parameters](#).

Here is an example of a [Components] section:

```
[Components]
Name: "main"; Description: "Main Files"; Types: full compact custom; Flags:
fixed
Name: "help"; Description: "Help Files"; Types: full
Name: "help\english"; Description: "English"; Types: full
Name: "help\dutch"; Description: "Dutch"; Types: full
```

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*)

#### *Description:*

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 `Components` 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*)

#### *Description:*

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

#### *Example:*

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

### **Types**

#### *Description:*

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**

*Description:*

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

*Example:*

ExtraDiskSpaceRequired: 0

## Flags

*Description:*

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 `Components` 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 AlwaysRestart 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 [Components and Tasks Parameters](#).

Here is an example of a [Tasks] section:

```
[Tasks]
Name: desktopicon; Description: "Create a &desktop icon"; GroupDescription:
"Additional icons:"; Components: main
Name: desktopicon\common; Description: "For all users"; GroupDescription:
"Additional icons:"; Components: main; Flags: exclusive
Name: desktopicon\user; Description: "For the current user only";
GroupDescription: "Additional icons:"; Components: main; Flags: exclusive
unchecked
Name: quicklaunchicon; Description: "Create a &Quick Launch icon";
GroupDescription: "Additional icons:"; Components: main; Flags: unchecked
Name: associate; Description: "&Associate files"; GroupDescription: "Other
tasks:"; Flags: unchecked
```

The following is a list of the supported [parameters](#):

### **Name** *(Required)*

#### *Description:*

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)*

#### *Description:*

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**

#### *Description:*

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**

*Description:*

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

*Description:*

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 same application is already installed. This flag cannot be combined with the `unchecked` flag.

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 AlwaysRestart but per task.

**unchecked**

Instructs Setup that this task should be unchecked initially. This flag cannot be combined with the `checkedonce` flag.

*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*)

#### *Description:*

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

#### *Example:*

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

### **Attribs**

#### *Description:*

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:*

```
Attribs: hidden system
```

### **Permissions**

#### *Description:*

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 user must be running Windows 2000 or later (NT 4.0 is not supported due to API bugs), 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: authusers-modify
```

**Flags**

*Description:*

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.

**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.

*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: onlyifdoesntexist
uninsneveruninstall
Source: "MYPROG.EXE"; DestDir: "{app}"
Source: "MYPROG.HLP"; DestDir: "{app}"
Source: "README.TXT"; DestDir: "{app}"; Flags: isreadme
```

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)

#### *Description:*

The name of the *source file*. The compiler will prepend the path of your installation's source directory 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)

#### *Description:*

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

#### *Description:*

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

#### *Description:*

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 "?").

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" would only exclude a file named "foo" at the base of the tree. On the other hand, "foo" would exclude any file named "foo" anywhere in the tree.

The patterns may include backslashes. "foo\bar" would exclude both "foo\bar" and "subdir\foo\bar". "\foo\bar" would only exclude "foo\bar".

*Examples:*

Source: "\*"; Excludes: "\*.\*"

Source: "\*"; Excludes: "\*.\*, \Temp\\*"; Flags: recursesubdirs

## CopyMode

*Description:*

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 -> **no flags**

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

*Description:*

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

*Description:*

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 user must be running Windows 2000 or later (NT 4.0 is not supported due to API bugs), 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: authusers-modify
```

**FontInstall***Description:*

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 on Windows 2000/XP/2003, the user must be a member of the Power Users or Administrators groups. On Windows NT 4.0 and earlier, anyone can install a font.

*Example:*

```
Source: "OZHANDIN.TTF"; DestDir: "{fonts}"; FontInstall: "Oz Handicraft  
BT"; Flags: onlyifdoesntexist uninsneveruninstall
```

**Flags***Description:*

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

**allowunsafe files**

Disables the compiler's automatic checking for unsafe files. 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:

1. Neither the existing file nor the file being installed has version info.
2. The `ignoreversion` flag is also used on the entry.
3. 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 be 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. This flag is useful if the file is handled by the `[Code]` section exclusively.

**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.

**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 be 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 `ExtractTemporaryFile` 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 OLE server (a.k.a. ActiveX control). With this flag set, Setup will locate and execute the DLL/OCX's `DllRegisterServer` export. The uninstaller calls `DllUnregisterServer`. When used in combination with `sharedfile`, the DLL/OCX will only be unregistered when the reference count reaches zero.

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.

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**

This flag is generally useful when replacing core system files. If the file existed beforehand and was found to be locked resulting in Setup being unable to replace it, Setup will register the file (either in `WININIT.INI` or by using `MoveFileEx`, for Windows and Windows NT respectively) to be replaced the next time the system is restarted. When this happens, the user will be prompted to restart the computer at the end of the installation process.

To maintain compatibility with Windows 95/98/Me, long filenames should not be used on an entry

with this flag. Only "8.3" filenames are supported. (Windows NT platforms do not have this limitation.)

**IMPORTANT:** The *restartreplace* flag will only successfully replace an in-use file on Windows NT platforms if the user has administrative privileges. If the user does not have administrative privileges, this message will be displayed: "RestartReplace failed: MoveFileEx failed; code 5." Therefore, when using *restartreplace* it is highly recommended that you have your installation require administrative privileges by setting "PrivilegesRequired=admin" in the [Setup] section.

#### **sharedfile**

Uses Windows' shared file counting feature (located in the registry at HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\SharedDLLs). This enables a file to be shared between applications, without worrying about it being inadvertently removed. Each time the file is installed, the *reference count* for the file is incremented. 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).

Most files installed to the Windows System directory should use this flag, including .OCX, .BPL, and .DPL files.

#### **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.

#### **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 SolidCompression is also used.

#### **touch**

This flag causes Setup to set the time/date stamp of the installed file(s) to that which is specified by the TouchDate 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 on Windows NT/2000/XP/2003 for this flag to have an effect.

#### **uninsneveruninstall**

Never uninstall 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.

*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 the `restartreplace` flag is not used and 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.

## [Icons] 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}\MYPROG.EXE"; WorkingDir:
"{app}"
Name: "{group}\Uninstall My Program"; Filename: "{uninstallexe}"
```

The following is a list of the supported parameters:

### **Name** (Required)

#### *Description:*

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: "{userdesktop}\My Program"
Name: "{commonprograms}\My Program"
Name: "{commonstartup}\My Program"
```

### **Filename** (Required)

#### *Description:*

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

#### *Examples:*

```
Filename: "{app}\MYPROG.EXE"
Filename: "{uninstallexe}"
```

### **Parameters**

#### *Description:*

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

#### *Example:*

```
Parameters: "/play filename.mid"
```

### **WorkingDir**

#### *Description:*

The working (or *Start In*) directory for the shortcut, which is the directory in which the program is started from. If this parameter is not specified or is blank, Windows will use a default path, which varies between the different Windows versions. This parameter can include constants.

#### *Example:*

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

### **HotKey**

#### *Description:*

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

*Description:*

Specifies the *Comment* (or "description") field of the shortcut, which determines the popup hint for it in Windows 2000, Me, and later. Earlier Windows versions ignore the comment.

*Example:*

```
Comment: "This is my program"
```

## IconFilename

*Description:*

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"
```

## IconIndex

*Default:*

0

*Description:*

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

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
```

## Flags

*Description:*

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.

### **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.

Folder shortcuts are only supported by Windows 2000, Me, and later. On earlier versions of Windows, Setup will fall back to creating a normal shortcut when this flag is used.

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.

**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\CurrentVersion\App 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: "{win}\MYPROG.INI"; Section: "InstallSettings"; Flags:
uninsdeletesection
Filename: "{win}\MYPROG.INI"; Section: "InstallSettings"; Key:
"InstallPath"; String: "{app}"
```

The following is a list of the supported parameters:

### Filename *(Required)*

*Description:*

The name of the .INI file you want Setup to modify, which can include constants. If this parameter is blank, it writes to WIN.INI in the system's Windows directory.

*Example:*

```
Filename: "{win}\MYPROG.INI"
```

### Section *(Required)*

*Description:*

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

*Example:*

```
Section: "Settings"
```

### Key

*Description:*

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

*Description:*

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

*Description:*

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 name doesn't already exist.

**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:

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 none is found, it searches for just a primary language identifier match. If two or more available languages have the same primary language identifier, it goes with the first one listed in the [Languages] section.
3. If none is found, it defaults to the first language specified in the [Languages] section.

If the `ShowLanguageDialog` [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.

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: "Dutch.isl"
```

### **Name** (Required)

#### *Description:*

The internal name of the language, which you can set to anything you like. This can be 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)

#### *Description:*

Specifies the name(s) of file(s) to read the default messages from. The file(s) must be located in your installation's source directory 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:MyMessages.isl"
```

### **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 source directory 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**

*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 source directory 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**

*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 source directory 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 [Constants](#) 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:CreateDesktopIcon}"
```

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 het &bureaublad
```

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 extension
AssociatingFileExtension=Associating %1 with the %2 file extension...
```

You may use these predefined custom messages in your own script. An example which uses UninstallProgram:

```
[Icons]
Name: "{group}\{cm:UninstallProgram,My Program}"; Filename:
"{uninstallexe}"
```

## [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
WelcomeFontName=Verdana
WelcomeFontSize=12
TitleFontName=Arial
TitleFontSize=29
CopyrightFontName=Arial
CopyrightFontSize=8
```

`LanguageName` is the name of the language. 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 on NT-based platforms, displayed as such). To embed Unicode characters, use "<nnnn>", where "nnnn" is the 4-digit hexadecimal Unicode character code. You can find Unicode character codes of characters using the Character Map accessory included with Windows 2000 and later.

`LanguageID` is the numeric "language identifier" of the language. See [http://msdn.microsoft.com/library/en-us/intl/nls\\_238z.asp](http://msdn.microsoft.com/library/en-us/intl/nls_238z.asp) for a list of valid language identifiers. This 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.

`LanguageCodePage` specifies the "code page" needed to display the language. When populating the list of available languages on the *Select Language* dialog in a multilingual installation, it compares the `LanguageCodePage` values against the system code page to determine which languages should be listed. Only languages whose `LanguageCodePage` values match the system code page are shown. 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 listed, 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.

`DialogFontName` and `DialogFontSize` specify the font name and point size to use in dialogs. 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, 8-point *Microsoft Sans Serif* or *MS Sans Serif* will be substituted.

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: uninsdeletekeyifempty
Root: HKCU; Subkey: "Software\My Company\My Program"; Flags: uninsdeletekey
Root: HKLM; Subkey: "Software\My Company"; Flags: uninsdeletekeyifempty
Root: HKLM; Subkey: "Software\My Company\My Program"; Flags: uninsdeletekey
Root: HKLM; Subkey: "Software\My Company\My Program"; ValueType: string;
ValueName: "InstallPath"; ValueData: "{app}"
```

The following is a list of the supported parameters:

### Root (Required)

#### *Description:*

The root key. This must be one of the following:

HKCR	(HKEY_CLASSES_ROOT)
HKCU	(HKEY_CURRENT_USER)
HKLM	(HKEY_LOCAL_MACHINE)
HKU	(HKEY_USERS)
HKCC	(HKEY_CURRENT_CONFIG)

#### *Example:*

```
Root: HKCU
```

### Subkey (Required)

#### *Description:*

The subkey name, which can include constants.

#### *Example:*

```
Subkey: "Software\My Company\My Program"
```

### ValueType

#### *Description:*

The data type of the value. This must be one of the following:

```
none
string
expandsz
multisz
dword
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. This data type is primarily used on Windows NT/2000/XP/2003, but is supported by Windows 95/98/Me.

If `multisz` is specified, Setup will create a multi-string (REG\_MULTI\_SZ) value.

If `dword` is specified, Setup will create an integer (REG\_DWORD) value.

If `binary` is specified, Setup will create a binary (REG\_BINARY) value.

*Example:*

ValueType: string

### ValueName

*Description:*

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

*Description:*

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`, 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

*Description:*

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 user must be running Windows 2000 or later (NT 4.0 is not supported due to API bugs) and 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.

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: authusers-modify
```

**Flags***Description:*

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.

**preserverstringtype**

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 `RestartIfNeededByRun` 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 the README file"; Flags:
postinstall shellexec skipifsilent
Filename: "{app}\MYPROG.EXE"; Description: "Launch application"; Flags:
postinstall nowait skipifsilent unchecked
```

The following is a list of the supported parameters:

### Filename *(Required)*

#### *Description:*

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

#### *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

#### *Description:*

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

#### *Example:*

```
Parameters: "/x"
```

### WorkingDir

#### *Description:*

The directory in which the program is started from. 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

*Description:*

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..."
```

## RunOnceId

*Description:*

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"
```

## Flags

*Description:*

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

### 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 an [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.

### 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 `.hlp`, `.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.

**skipifnotsilent**

Valid only in an [Run] section. Instructs Setup to skip this entry if Setup is not running (very) silent.

**skipifsilent**

Valid only in an [Run] section. Instructs Setup to skip this entry if Setup is running (very) silent.

**unchecked**

Valid only in an [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 an example of a [UninstallDelete] section:

```
[UninstallDelete]
Type: files; Name: "{win}\MYPROG.INI"
```

The following is a list of the supported parameters:

### Type (Required)

#### Description:

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)

#### Description:

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. I strongly recommend against doing this 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

## **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 <http://www.jrsoftware.org/isfaq.php>

## Wizard Pages

Below is a list of all the wizard pages Setup may potentially display, and the conditions under which they are displayed.

- **Welcome**  
Always shown.
- **License Agreement**  
Shown if LicenseFile is set. Users may proceed to the next page only if the option "I accept the agreement" is selected.
- **Password**  
Shown if Password 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 UserInfoPage is set to *yes*.
- **Select Destination Location**  
Shown by default, but can be disabled via DisableDirPage.
- **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 DisableProgramGroupPage.
- **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 DisableReadyPage.
- **Preparing to Install**  
Normally, Setup will never stop on this page. The only time it will is if Setup determines it can't continue. Currently, the only time this can happen is if one or more files specified in the [Files] section 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.
- **Installing**  
Shown during the actual installation process.
- **Information**  
Shown if InfoAfterFile is set.
- **Setup Completed**  
Shown by default, but can be disabled in some cases via DisableFinishedPage.

## 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.)
- [Icons] 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 ChangesAssociations was set to `yes`, file associations are refreshed now.
- If ChangesEnvironment 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: "InstallSettings"; Flags:
uninsdeletesectionifempty
Filename: "{win}\MYPROG.INI"; Section: "InstallSettings"; Key: "InstallPath";
String: "{app}"; Flags: uninsdeleteentry
```

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]` entries in the same order they appear in the script (not in reverse order).

## Miscellaneous Notes

- If Setup detects a shared version of Windows on the user's system where the Windows System directory is write protected, the {sys} directory constant will translate to the user's Windows directory instead of the System directory.
- 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=c:\Program Files\My Program"
```

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 when it is silent or very silent.

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 specify an output path (overriding any `OutputDir` setting in the script), "/F" to specify an output filename (overriding any `OutputBaseFilename` setting in the script), "/Q" for quiet compile (print only error messages), and "/"?" to show a help screen.

Example:

```
iscc /Q /O"My Output" /F"MyProgram-1.0" "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.

### **/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'` and `'/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/NoUninstallWarning/ExitSetupMessage/ConfirmUninstall` 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**

Instructs Setup not to reboot even if it's necessary.

**/RESTARTEXITCODE=*exit code***

Specifies the custom exit code that Setup is to return when a restart is needed. Useful along with '/NORESTART'. Also see [Setup Exit Codes](#).

**/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.

**/GROUP="*folder name*"**

Overrides the default folder name displayed on the *Select Start Menu Folder* wizard page. 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 any icons* check box on the *Select Start Menu Folder* wizard page.

**/COMPONENTS="*comma separated list of component names*"**

Overrides the default components settings. Using this command line parameter causes Setup to automatically select a custom type.

**/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.
- 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).

Before returning an exit code of 1, 3, or 4, 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 [Setup Command Line Parameters](#) 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.) Currently, it is not possible to customize the filename.

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.

### **/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 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, and were you not to notice this and take action, it could also lead to 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.

### COMCAT.DLL version 5.0

Version 5.0 of COMCAT.DLL must not be redistributed because it does not work on Windows 95 or NT 4.0. If you need to install COMCAT.DLL, use version 4.71 instead.

Reference: <http://support.microsoft.com/support/kb/articles/Q201/3/64.ASP>

### 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 download the COMCTL32 update from Microsoft, or distribute the COMCTL32 update along with your program.

Reference: <http://www.microsoft.com/permission/copyrgt/cop-soft.htm#COM>

Reference: <http://www.microsoft.com/downloads/details.aspx?FamilyID=cb2cf3a2-8025-4e8f-8511-9b476a8d35d2&DisplayLang=en>

### CTL3D32.DLL, Windows NT-specific version

Previously, on the "Installing Visual Basic 5.0 & 6.0 Applications" How-To page there was a version of CTL3D32.DLL included in the zip files. At the time I included it, I was not aware that it only was compatible with Windows NT. Now if you try to install that particular version of CTL3D32.DLL you must use a `MinVersion` setting that limits it to Windows NT platforms only. (You shouldn't need to

install CTL3D32.DLL on Windows 95/98/Me anyway, since all versions have a 3D look already.)

**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.

## 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 zlib compression library that Inno Setup uses.

Julian Seward: Creator of the bzlib compression library that Inno Setup uses.

Igor Pavlov: Creator of the 7-Zip LZMA compression library that Inno Setup uses.

?: Most of the disk spanning code (1.09). (Sorry, I somehow managed to lose your name!)

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 appending to existing uninstall logs (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); Various other 4.0.0+ features.

Alex Yackimoff: Portions of TNewCheckListBox (4.0.0).

Carlo Kok: RemObjects Pascal Script (4.0.0).

Creators of SynEdit: The syntax-highlighting editor used in the Compiler (2.0.0).

glyFX: The Inno Setup logo, the compiler icon, the document icon, the Inno Setup installer wizard images and the images for the IDE's toolbar.

If I have left anyone out, please don't hesitate to let me know.

## **Contacting Me**

The latest versions of Inno Setup and other software I've written can be found on my web site at:  
<http://www.jrsoftware.org/>

For information on contacting me and obtaining technical support for Inno Setup, go to this page:  
<http://www.jrsoftware.org/contact.php>

## **[Setup]: UseSetupLdr**

**Valid values:** yes or no

**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.

## **[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:** yes or no

**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]: AppName**

### **Description:**

This required directive specifies the title of the application you are installing. Do not include the version number, as the `AppVerName` directive is for that purpose. `AppName` is shown throughout the installation process, in places like the upper-left corner of the Setup screen, and in the wizard. The value may include constants.

**Example:**     `AppName=My Program`

## **[Setup]: AppVerName**

### **Description:**

The value of this required directive should be the same (or similar to) the value of AppName, but it should also include the program's version number. The value may include constants.

**Example:**     AppVerName=My Program version 3.0

## [Setup]: AppId

**Default value:** AppName

### **Description:**

The value of `AppId` is stored inside uninstall log files (`unins???.dat`), and is checked by subsequent installations to determine whether it may append to a particular existing uninstall log. 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 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 UsePreviousAppDir). 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]: 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."

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 "kernel32" _  
    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*.

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

**Example:**     AppMutex=MyProgramsMutexName

## **[Setup]: AppCopyright**

### **Description:**

This is optional, and is only used to display a copyright message in the bottom-right corner of Setup's background window.

Note that the copyright message will only be seen if WindowVisible is *yes*.

**Example:**     AppCopyright=Copyright © 1997 My Company, Inc.

**[Setup]: AppComments, AppContact, AppPublisher, AppPublisherURL, AppReadmeFile, AppSupportURL, AppUpdatesURL, AppVersion**

**Description:**

These are all used for display purposes on the "Support" dialog of the *Add/Remove Programs* Control Panel applet in Windows 2000/XP/2003. Setting them is optional, and will have no effect on earlier Windows versions. The values may include constants.

**Example:**

```
AppPublisher=My Company, Inc.  
AppPublisherURL=http://www.mycompany.com/  
AppVersion=1.5
```

## **[Setup]: DefaultDirName**

### **Description:**

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

### **Example:**

#### **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]: Uninstallable**

**Valid values:** yes or no

**Default value:** `yes`

### **Description:**

This determines if Inno Setup's automatic uninstaller is to be included in the installation. If this is `yes` the uninstaller is included. If this is `no`, no uninstallation support is included, requiring the end-user to manually remove the files pertaining to your application.

## **[Setup]: MinVersion**

**Format:** *a.bb, c.dd*, where *a.bb* is the Windows version, and *c.dd* is the Windows NT version.

**Default value:** 4.0, 4.0

### **Description:**

This directive lets you specify a minimum version of Windows or Windows NT that your software runs on. To prevent your program from running on Windows or Windows NT, specify "0" for one the minimum versions. Build numbers and/or service pack levels may be included in the version numbers.

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

## **[Setup]: OnlyBelowVersion**

**Format:** `a.bb, c.dd`, where `a.bb` is the Windows version, and `c.dd` is the Windows NT version.

**Default:** `0,0`

### **Description:**

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

This directive is essentially the opposite of MinVersion.

## **[Setup]: AdminPrivilegesRequired**

**Valid values:** yes or no

**Default value:** no

### **Description:**

*Obsolete in 3.0.4.* While `AdminPrivilegesRequired` is still recognized by the compiler, `PrivilegesRequired` supersedes and overrides it.

When set to `yes`, Setup will give an error message at startup ("You must be logged in as an administrator when installing this program") if the user doesn't have administrative privileges. This only applies to Windows NT platforms.

## **[Setup]: PrivilegesRequired**

**Valid values:** none, poweruser, or admin

**Default value:** none

### **Description:**

This directive specifies the minimum user privileges required to run the installation. When set to `poweruser` or `admin`, Setup will give an error message at startup (e.g. "You must be logged in as an administrator when installing this program") if the user doesn't have at least Power User or administrative privileges, respectively. This only applies to Windows NT platforms.

## **[Setup]: DisableAppendDir**

**Valid values:** yes or no

**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]: 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]: 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 [\[Icons\]](#) section entry.

## **[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]: Compression

**Valid values:** zip  
zip/1 through zip/9  
bzip  
bzip/1 through bzip/9  
lzma  
lzma/fast  
lzma/normal  
lzma/max  
lzma/ultra (read warning below before using ultra)  
none

**Default value:** lzma

### 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` has one unique property, though: unlike the other methods, 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 `bzip2` 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.

`lzma` is the method of compression employed by the `7-Zip LZMA` 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 `lzma` compression levels. If a compression level isn't specified, it defaults to `max`.

	<u>Decompression</u>	<u>Compression</u>	
fast	3 MB	3 MB	
normal	4 MB	27 MB	
max (default)	10 MB	84 MB	
ultra	34 MB	369 MB	<-- that's not a typo; be careful!

`none` specifies that no compression be used.

### See also:

[SolidCompression](#)

## [Setup]: SolidCompression

**Valid values:** yes or no

**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. Be sure to also use `Compression=lzma` or `Compression=bzip`, since the default `zip` compression doesn't benefit too much from solid compression (as it works with smaller blocks).

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]: 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]: CreateAppDir**

**Valid values:** yes or no

**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:** yes or no

**Default value:** *yes*

### **Description:**

If this is set to *no*, Setup won't create an entry in the *Add/Remove Programs* Control Panel applet. This 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*).

When this directive is set to *no*, UpdateUninstallLogAppName is usually set to *no* as well.

## **[Setup]: DirExistsWarning**

**Valid values:** auto, yes, or no

**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 same application 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]: DisableDirPage**

**Valid values:** yes or no

**Default value:** no

### **Description:**

If this is set to `yes`, Setup will not show the *Select Destination Location* wizard page. In this case, it will always use the default directory name.

## **[Setup]: DisableFinishedPage**

**Valid values:** yes or no

**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:** yes or no

**Default value:** no

### **Description:**

If this is set to `yes`, Setup will not show the *Select Start Menu Folder* wizard page. In this case, it uses the folder name specified by the `DefaultGroupName` [Setup] section directive, or "(Default)" if none is specified.

## **[Setup]: DisableReadyMemo**

**Valid values:** yes or no

**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:** yes or no

**Default value:** no

### **Description:**

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

## [Setup]: UserInfoPage

**Valid values:** yes or no

**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 `{userinfo}`, `{userinfoorg}` and `{userinfoserial}` constants. You can use these constants in [Registry] or [INI] entries to save their values for later use.

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

**[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]: AlwaysUsePersonalGroup**

**Valid values:** yes or no

**Default value:** no

### **Description:**

Normally on Windows NT platforms, 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]: 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]: 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 append to them.

**Example:**      UninstallFilesDir={app}\uninst

## **[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 on Windows 2000/XP/2003. 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]: 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 [\[Icons\]](#) section entry.

## [Setup]: UninstallLogMode

**Valid values:** append, new, or overwrite

**Default value:** append

### **Description:**

`append`, the default setting, instructs Setup to append to an existing uninstall log 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 same application 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]: 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]: 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]: 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]: DiskSpanning

**Valid values:** yes or no

**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 DiskSliceSize and SlicesPerDisk [Setup] section directives, respectively. Other disk spanning-related directives that you may want to tweak include DiskClusterSize and ReserveBytes.

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]: DiskSliceSize**

**Valid values:** 262144 through 2100000000

**Default value:** 1457664 (the size of a 1.44MB floppy)

### **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 700 MB (80-minute) recordable CDs, use:

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

## **[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]: 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]: 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]: MergeDuplicateFiles**

**Valid values:** yes or no

**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]: DontMergeDuplicateFiles**

**Valid values:** yes or no

### **Description:**

*Obsolete in 4.2.5. Use MergeDuplicateFiles instead.*

MergeDuplicateFiles=no is equivalent to DontMergeDuplicateFiles=yes.

## **[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]: AllowNoIcons**

**Valid values:** yes or no

**Default value:** no

### **Description:**

This is used to determine whether Setup should display a *Don't create any icons* check box, which allows the user to skip creation of program icons. If it is no the check box will not be displayed; if it is yes it will be displayed.

## **[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:** yes or no

**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.

## **[Setup]: AlwaysRestart**

**Valid values:** yes or no

**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]: RestartIfNeededByRun**

**Valid values:** yes or no

**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]: MessagesFile**

### **Description:**

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

## **[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 source directory 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]: 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 source directory 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]: 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 source directory 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]: 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.

On Windows NT platforms, 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]: UsePreviousAppDir**

**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 same application 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.

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.

## [Setup]: UsePreviousGroup

**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 same application 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 any icons* setting from the previous installation will be restored.

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.

## **[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 same application 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.

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.

## **[Setup]: UsePreviousTasks**

**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 same application 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.

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`.

## **[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 same application 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.

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`.

## **[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 Encryption 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 128-bit MD5 hash, salted with a 64-bit random number. (Note: When encrypted is enabled, this stored hash is *not* used for the encryption key; a different hash with a different salt is generated for that.)

## **[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 source directory 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]: WindowShowCaption**

**Valid values:** yes or no

**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]: WindowResizable**

**Valid values:** yes or no

**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]: WindowVisible**

**Valid values:** yes or no

**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 WizardImageFile). There can only be unused space if WizardImageStretch is set to `no`.

## **[Setup]: WizardImageStretch**

**Valid values:** yes or no

**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 WizardImageStretch 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]: SourceDir**

### **Description:**

Specifies a new source directory for the script.

**Example:**      SourceDir=c:\files

## **[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`. Setting `OutputDir` to `.` will result in the files being placed in the source directory.

**Example:**      `OutputDir=c:\output`

## **[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.

## **[Setup]: UninstallStyle**

### **Description:**

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

## **[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 source directory 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]: 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:** yes or no

**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]: 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]: 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]: 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]: ShowLanguageDialog**

**Valid values:** `yes`, `no`, 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.

## **[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 since Windows 2000 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]: 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:** yes or no

**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]: SetupIconFile**

### **Description:**

Specifies a custom program icon to use for Setup/Uninstall. The file must be located in your installation's source directory 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 directive is not supported if you're compiling under Windows 95/98/Me.

**Example:**      SetupIconFile=MyProgSetup.ico

**[Setup]: VersionInfoCompany**

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

**Description:**

Specifies the company name value for the Setup version info.

**[Setup]: VersionInfoDescription**

**Default value:** "AppName Setup" if AppName doesn't include constans, an empty string otherwise

**Description:**

Specifies the file description value for the Setup version info.

**[Setup]: VersionInfoTextVersion**

**Default value:** VersionInfoVersion

**Description:**

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

**[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]: 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]: 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]: 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 `TimeStampsInUTC` 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 `TimeStampsInUTC` is set to `yes`, the time is assumed to be a UTC time.

**Example:** `TouchTime=13:00`

## [Setup]: Encryption

**Valid values:** `yes or no`

**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 128-bit key derived from the value of the Password [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 [Inno Setup Downloads](#) page for more information.

If encryption is enabled and you call the `ExtractTemporaryFile` 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 128-bit MD5 hash of 64-bit random salt plus the value of Password.

## **[Setup]: AppModifyPath**

### **Description:**

When this directive is set, a separate "Modify" button in the Add/Remove Programs Control Panel applet in Windows 2000/XP/2003 will be displayed. Setting it is optional, and will have no effect on earlier Windows versions. The value may include constants.

**Example:**      `AppModifyPath="{app}\Setup.exe" /modify=1`

## **[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 output directory unless a path is included.

**Example:**      `OutputManifestFile=Setup-Manifest.txt`

## **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`.

## 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 same application 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 messages 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 AppName from the most recent installation, unless UpdateUninstallLogAppName 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 UninstallLogMode.

*Note:* Setup can only append to uninstall log files that were created by an Inno Setup 1.3.1 (or later) installation.

## Same Application

"Same application" refers to two separate installations that share the same AppId setting (or if AppId is not set, the same AppName setting).

## 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 SourceDir directive in the script's [Setup] section.

## **Using Build Number and/or Service Pack Levels**

The version numbers in `MinVersion` and `OnlyBelowVersion` can include build numbers and/or service pack levels. Examples: `5.0.2195`, `5.0sp1`, `5.0.2195sp1`. If a build number is not specified or is zero, Setup will not check the build number. If a service pack level is not specified or is zero, Setup interprets it as meaning "no service pack."

## Windows Versions

Windows versions:

4.0.950	Windows 95
4.0.1111	Windows 95 OSR 2 & OSR 2.1
4.0.1212	Windows 95 OSR 2.5
4.1.1998	Windows 98
4.1.2222	Windows 98 Second Edition
4.9.3000	Windows Me

Windows NT versions:

4.0.1381	Windows NT 4.0
5.0.2195	Windows 2000
5.01.2600	Windows XP
5.02.3790	Windows Server 2003, and Windows XP x64 Edition

Note that there is normally no need to specify the build numbers (i.e. you may simply use "4.1" for Windows 98).

## 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: 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 <http://www.remobjects.com/?ps> for more information.

### See also

[Creating the \[Code\] section](#)

[Event Functions](#)

[Scripted Constants](#)

[Check Parameters](#)

[BeforeInstall and AfterInstall Parameters](#)

[Uninstall Code](#)

[Examples](#)

[Support Functions Reference](#)

[Support Classes Reference](#)

[Using Custom Wizard Pages](#)

[Using DLLs](#)

[Using COM Automation objects](#)

## **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 <http://www.marcocantu.com/epascal/>.

## Pascal Scripting: Event functions

The Pascal script can contain several event functions which are called at appropriate times. For Setup these are:

- `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: TSetupStep);`  
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.
- `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 `wpWelcome`, `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(CurPageID: 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 MD5 hash of your password yourself and

then compare that to `GetMD5OfString (Password)`. This way the actual value of the password remains protected.

Note: if you have a `CheckPassword` event function and your users run Setup with both the `"/PASSWORD="` and `"/SILENT"` command line parameters set, your `CheckPassword` function will be called \*before\* any other event function is called, including `InitializeSetup`.

- `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 `SetPreviousData(PreviousDataKey, ...)` 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 Setup Exit Codes.

For Uninstall these are:

- `function InitializeUninstall(): Boolean;`  
Return False to abort Uninstall, True otherwise.
- `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.

Here's the list of constants used by these functions:

*CurStep values*  
`ssInstall, ssPostInstall, ssDone`

*CurUninstallStep values*

usAppMutexCheck, usUninstall, usPostUninstall, usDone

*CurPageID values for predefined wizard pages*

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 value.

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 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".
- *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 support function. Here is an example.

```
[INI]
FileName: "{app}\MyIni.ini"; Section: "MySettings"; Key: "ShortApp";
String: "{code:GetShortName|{app}}"
```

### **See also** [Constants](#)

## Pascal Scripting: Check Parameters

There is one optional parameter that is supported by all sections whose entries are separated into parameters. This is:

### Check

#### *Description:*

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 support function.

Besides a single name, you may also use boolean expressions. See Components and Tasks parameters 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.

There's one support function that may be called from within a parameter list: `ExpandConstant`.

#### *Example:*

```
[Files]
Source: "MYPROG.EXE"; DestDir: "{app}"; Check: MyProgCheck
Source: "A\MYFILE.TXT"; DestDir: "{app}"; Check:
MyDirCheck(ExpandConstant('{app}\A'))
Source: "B\MYFILE.TXT"; DestDir: "{app}"; Check:
DirExists(ExpandConstant('{app}\B'))
```

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.

Here is an example of a [Code] section containing the check functions used above. Function `DirExists` is a support function 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 install MyProg.exe to ' +
      ExtractFilePath(CurrentFileName) + '?', mbConfirmation, MB_YESNO) = idYes;
    MyProgChecked := True;
  end;
  Result := MyProgCheckResult;
end;

function MyDirCheck(DirName: String): Boolean;
begin
  Result := DirExists(DirName);
end;
```



## Pascal Scripting: BeforeInstall and AfterInstall Parameters

There are two optional parameters that are supported by all sections whose entries are separated into parameters except for [Types], [Components] and [Tasks]. These are:

### BeforeInstall

*Description:*

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 support function.

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.

There's one support function that may be called from within a parameter list: ExpandConstant.

*Example:*

```
[Files]
Source: "MYPROG.EXE"; DestDir: "{app}"; BeforeInstall: MyBeforeInstall
Source: "A\MYFILE.TXT"; DestDir: "{app}"; BeforeInstall:
MyBeforeInstall2('{app}\A\MYFILE.TXT')
Source: "B\MYFILE.TXT"; DestDir: "{app}"; BeforeInstall:
MyBeforeInstall2('{app}\B\MYFILE.TXT')
Source: "MYPROG.HLP"; DestDir: "{app}"; BeforeInstall: Log('Before
MYPROG.HLP Install')
```

### AfterInstall

*Description:*

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 support function. May include one parameter that Setup should pass to the function. This parameter may include constants.

*Example:*

```
[Files]
Source: "MYPROG.EXE"; DestDir: "{app}"; AfterInstall: MyAfterInstall
Source: "A\MYFILE.TXT"; DestDir: "{app}"; AfterInstall:
MyAfterInstall2('{app}\A\MYFILE.TXT')
Source: "B\MYFILE.TXT"; DestDir: "{app}"; AfterInstall:
MyAfterInstall2('{app}\B\MYFILE.TXT')
Source: "MYPROG.HLP"; DestDir: "{app}"; AfterInstall: Log('After MYPROG.HLP
Install')
```

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.

Here is an example of a [Code] section containing the functions used above. Function Log is a support function and therefore not included in this [Code] section.

```
[Code]
procedure MyBeforeInstall();
begin
  MsgBox('About to install MyProg.exe as ' + CurrentFileName + '.',
mbInformation, MB_OK);
end;

procedure MyBeforeInstall2(FileName: String);
begin
  MsgBox('About to install ' + FileName + ' as ' + CurrentFileName + '.',
mbInformation, MB_OK);
end;

procedure MyAfterInstall();
begin
  MsgBox('Just installed MyProg.exe as ' + CurrentFileName + '.',
mbInformation, MB_OK);
end;

procedure MyAfterInstall2(FileName: String);
begin
  MsgBox('Just installed ' + FileName + ' as ' + CurrentFileName + '.',
mbInformation, MB_OK);
end;
```

## Pascal Scripting: Uninstall Code

The Pascal script can also contain code invoked at uninstall time. See the [Event Functions](#) 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: 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 [Support Functions](#) 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 [Support Functions](#) 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> <callingconvention>';

function A(B: Integer): Integer;
external '<dllfunctionname>@<dllfilename> <callingconvention> <options>';
```

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 exception when you try to call the DLL function which you can catch to handle the absence of the DLL function.

### **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 'MyDll.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 to instruct Setup to automatically extract the DLL from the [Files] section may also be used. For example:

```
[Files]
Source: "MyDll.dll"; Flags: dontcopy

[Code]
procedure MyDllFunc(hWnd: Integer; lpText, lpCaption: String; uType:
Cardinal);
external 'MyDllFunc@files:MyDll.dll stdcall';
```

Open the "CodeDll.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.

There are two support functions related to creating COM Automation objects: CreateOleObject and GetActiveOleObject.

Use CreateOleObject 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. The returned value can then be used to access the methods and properties 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.

Use GetActiveOleObject 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.

Open the "CodeAutomation.iss" file in the "Examples" subdirectory in your Inno Setup directory for an example script using COM Automation objects.

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 CoFreeUnusedLibraries. This Windows function will then attempt to unload the library so you can delete it.



## **Pascal Scripting: Support Functions Reference**

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 ActiveLanguage: String;

function SetupMessage(const ID: TSetupMessageID): String;

function WizardDirValue: String;
function WizardGroupValue: String;
function WizardNoIcons: Boolean;
function WizardSetupType(const Description: Boolean): String;
function WizardSelectedComponents(const Descriptions: Boolean): String;
function WizardSelectedTasks(const Descriptions: Boolean): String;
function WizardSilent: Boolean;

function IsUninstaller: Boolean;
function UninstallSilent: Boolean;

function CurrentFileName: 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 GetPreviousData(const ValueName, DefaultValueData: String):
String;
function SetPreviousData(const PreviousDataKey: Integer; const ValueName,
ValueData: String): Boolean;

function Terminated: Boolean;
```

## Exception functions

```
procedure Abort;
procedure RaiseException(const Msg: String);

function GetExceptionMessage: String;
procedure ShowExceptionMessage;
```

## System functions

```
function IsAdminLoggedOn: Boolean;
function IsPowerUserLoggedOn: Boolean;
function UsingWinNT: Boolean;
function GetWindowsVersion: Cardinal;
function GetWindowsVersionString: String;

function InstallOnThisVersion(const MinVersion, OnlyBelowVersion: String):
```

```

Integer;

function GetEnv(const EnvVar: String): String;
function GetUserNameString: String;
function GetComputerNameString: String;

function GetUILanguage: Integer;

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 CreateMutex(const Name: String);
function CheckForMutexes(Mutexes: String): Boolean;

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; Indx, Count: Integer): String;
function Length(s: String): Longint;
function Lowercase(s: string): String;
function StringOfChar(c: Char; I : Longint): String;
function Uppercase(s: string): String;
procedure Delete(var S: String; Indx, Count: Integer);
procedure Insert(Source: String; var Dest: String; Indx: Integer);
procedure StringChange(var S: String; const FromStr, ToStr: String);
function Pos(SubStr, S: String): Integer;
function AddQuotes(const S: String): String;
function RemoveQuotes(const S: String): String;
function ConvertPercentStr(var S: String): Boolean;

function CompareText(const S1, S2: string): Integer;
function CompareStr(const S1, S2: string): Integer;

function Format1(const Format, S1: String): String;
function Format2(const Format, S1, S2: String): String;
function Format3(const Format, S1, S2, S3: String): String;
function Format4(const Format, S1, S2, S3, S4: String): String;

function Trim(const S: string): String;

```

```

function TrimLeft(const S: string): String;
function TrimRight(const S: string): String;

function StrToIntDef(s: string; def: Longint): Longint;
function StrToInt(s: string): Longint;
function IntToStr(i: Longint): String;

function CharLength(const S: String; const Index: Integer): Integer;

function AddBackslash(const S: String): String;
function RemoveBackslashUnlessRoot(const S: String): String;
function RemoveBackslash(const S: String): String;
function AddPeriod(const S: 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 GetDateTimeString(const DateTimeFormat: String; const
DateSeparator, TimeSeparator: Char): String;

procedure SetLength(var S: String; L: Longint);
procedure CharToOemBuff(var S: String);
procedure OemToCharBuff(var S: String);

function GetMD5OfString(const S: String): String;

function SysErrorMessage(ErrorCode: Integer): String;

```

### **Array functions**

```

function GetArrayLength(var Arr: Array): Longint;
procedure SetArrayLength(var Arr: Array; I: Longint);

```

### **File System functions**

```

function DirExists(const Name: String): Boolean;
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 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 GetCurrentDir: String;
function SetCurrentDir(const Dir: string): Boolean;
function GetWinDir: String;
function GetSystemDir: String;

```

```
function GetTempDir: String;
function GetShellFolder(Common: Boolean; const ID: TShellFolderID): String;
function GetShellFolderByCSIDL(const Folder: Integer; const Create:
Boolean): String;

function GetShortName(const LongName: String): 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 GetMD5OfFile(const Filename: String): String;
```

### **File functions**

```
function Exec(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 RenameFile(const OldName, NewName: string): Boolean;
function ChangeFileExt(const FileName, Extension: string): String;
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 LoadStringFromFile(const FileName: String; var S: String):
Boolean;
function LoadStringsFromFile(const FileName: String; var S:
TArrayOfString): Boolean;
function SaveStringToFile(const FileName, S: String; const Append:
Boolean): Boolean;
function SaveStringsToFile(const FileName: String; const S: TArrayOfString;
const Append: Boolean): 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;

procedure RegisterServer(const Filename: String; const FailCriticalErrors:
Boolean);
function UnregisterServer(const Filename: String; const FailCriticalErrors:
Boolean): Boolean;
procedure RegisterTypeLibrary(const Filename: String);
function UnregisterTypeLibrary(const Filename: String): Boolean
```

```
procedure IncrementSharedCount(const Filename: String; const
AlreadyExisted: Boolean);
function DecrementSharedCount(const Filename: String): Boolean;
procedure RestartReplace(const TempFile, DestFile: String);
procedure UnregisterFont(const FontName, FontFilename: String);
function ModifyPifFile(const Filename: String; const CloseOnExit: Boolean):
Boolean;
```

## Registry functions

```
function RegKeyExists(const RootKey: Integer; const SubKeyName: String):
Boolean;
```

```
function RegValueExists(const RootKey: Integer; const SubKeyName,
ValueName: String): Boolean;
```

```
function RegGetSubkeyNames(const RootKey: Integer; const SubKeyName:
String; var Names: TArrayOfString): Boolean;
```

```
function RegGetValueNames(const RootKey: Integer; const SubKeyName: String;
var Names: TArrayOfString): Boolean;
```

```
function RegQueryStringValue(const RootKey: Integer; const SubKeyName,
ValueName: String; var ResultStr: String): Boolean;
```

```
function RegQueryMultiStringValue(const RootKey: Integer; const SubKeyName,
ValueName: String; var ResultStr: String): Boolean;
```

```
function RegQueryDWordValue(const RootKey: Integer; const SubKeyName,
ValueName: String; var ResultDWord: Cardinal): Boolean;
```

```
function RegQueryBinaryValue(const RootKey: Integer; const SubKeyName,
ValueName: String; var ResultStr: String): 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, Data: String): Boolean;
```

```
function RegDeleteKeyIncludingSubkeys(const RootKey: Integer; const
SubkeyName: String): Boolean;
```

```
function RegDeleteKeyIfEmpty(const RootKey: Integer; const SubkeyName:
String): Boolean;
```

```
function RegDeleteValue(const RootKey: Integer; 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 SetIniBool(const Section, Key: String; const Value: Boolean; const  
Filename: String): Boolean;
```

```
function SetIniInt(const Section, Key: String; const Value: Longint; const  
Filename: String): Boolean;
```

```
function SetIniString(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 CreateInputFilePage(const AfterID: Integer; const ACaption,  
ADescription, ASubCaption: String): TInputFileWizardPage;
```

```
function CreateOutputMsgPage(const AfterID: Integer; const ACaption,  
ADescription, AMsg: String): TOutputMsgWizardPage;
```

```
function CreateOutputMsgMemoPage(const AfterID: Integer; const ACaption,  
ADescription, ASubCaption, AMsg: String): TOutputMsgMemoWizardPage;
```

```
function CreateOutputProgressPage(const ACaption, ADescription: String):  
TOutputProgressWizardPage;
```

```
function CreateCustomPage(const AfterID: Integer; const ACaption,  
ADescription: String): TWizardPage;
```

```
function CreateCustomForm: TSetupForm;
```

```
function PageFromID(const ID: Integer): TWizardPage;
```

```
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 BrowseForFolder(const Prompt: String; var Directory: String; const  
NewFolderButton: Boolean): Boolean;
```

```
function ExitSetupMsgBox: Boolean;
```

### **COM Automation objects support functions**

```
function CreateOleObject(const ClassName: string): Variant;
```

```
function GetActiveOleObject(const ClassName: string): Variant;
```

```
procedure CoFreeUnusedLibraries;
```

### **Setup Logging functions**

```
procedure Log(const S: String);
```

### **Other functions**

```
procedure Sleep(const Milliseconds: LongInt);  
function Random(const Range: Integer): Integer;  
procedure Beep;
```

```
procedure BringToFrontAndRestore;
```

### **Deprecated functions**

```
function LoadDLL(const DLLName: String; var ErrorCode: Integer): Longint;  
function CallDLLProc(const DLLHandle: Longint; const ProcName: String;  
const Param1, Param2: Longint; var Result: Longint): Boolean;  
function FreeDLL(const DLLHandle: Longint): Boolean;  
  
function CastStringToInteger(var S: String): Longint;  
function CastIntegerToString(const L: Longint): String;
```

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

*TGetShellFolderID*

sfDesktop, sfStartMenu, sfPrograms, sfStartup, sfSendTo, sfFonts,  
sfAppData, sfDocs, sfTemplates, sfFavorites, sfLocalAppData

*Reg\* - RootKey values*

HKEY\_CLASSES\_ROOT, HKEY\_CURRENT\_USER, HKEY\_LOCAL\_MACHINE, HKEY\_USERS,  
HKEY\_PERFORMANCE\_DATA, HKEY\_CURRENT\_CONFIG, HKEY\_DYN\_DATA,  
HKCR, HKCU, HKLM, HKU, HKCC

*TShouldProcessEntryResult*

srNo, srYes, srUnknown

*InstallOnThisVersion - return values*

irInstall, irNotOnThisPlatform, irVerTooLow, irVerTooHigh, irInvalid

*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

## **Support function: GetCmdTail**

### **Prototype:**

```
function GetCmdTail: String;
```

### **Description:**

Returns all command line parameters passed to Setup or Uninstall as a single string.

## **Support function: ParamCount**

### **Prototype:**

```
function ParamCount: Integer;
```

### **Description:**

Returns the number of command line parameters passed to Setup or Uninstall.

## **Support function: ParamStr**

### **Prototype:**

```
function ParamStr(Index: Integer): String;
```

### **Description:**

Returns the Index-th command line parameter passed to Setup or Uninstall.

## **Support function: ActiveLanguage**

### **Prototype:**

```
function ActiveLanguage: String;
```

### **Description:**

Returns the name of the active language.

## Support function: 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;
```

## Support function: 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.

## Support function: 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.

## **Support function: WizardNoIcons**

### **Prototype:**

```
function WizardNoIcons: Boolean;
```

### **Description:**

Returns the current setting of the *Don't create any icons* check box on the *Select Start Menu Folder* page of the wizard.

## **Support function: WizardSetupType**

### **Prototype:**

```
function WizardSetupType(const Description: Boolean): String;
```

### **Description:**

Returns the name or description of the setup type selected by the user.

## **Support function: 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.

## **Support function: 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.

## **Support function: WizardSilent**

### **Prototype:**

```
function WizardSilent: Boolean;
```

### **Description:**

Returns True if Setup is running silently, False otherwise.

## **Support function: IsUninstaller**

### **Prototype:**

```
function IsUninstaller: Boolean;
```

### **Description:**

Returns True if Uninstall is running as opposed to Setup, False otherwise.

## **Support function: UninstallSilent**

### **Prototype:**

```
function UninstallSilent: Boolean;
```

### **Description:**

Returns True if Uninstall is running silently, False otherwise.

## **Support function: CurrentFileName**

### **Prototype:**

```
function CurrentFileName: String;
```

### **Description:**

Returns the destination 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.

## Support function: 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.

## Support function: 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.

## Support function: 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 Components parameter.

### Example:

```
begin
  if IsComponentSelected('helpfiles') then
    // the 'helpfiles' component is selected
end;
```

## Support function: 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 Tasks parameter.

### Example:

```
begin
  if IsTaskSelected('desktopicon') then
    // the 'desktopicon' task is selected
end;
```

## Support function: 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.

### Example:

[Files]

```
Source: "Readme.txt"; Flags: dontcopy
```

[Code]

```
function InitializeSetup: Boolean;
```

```
var
```

```
    S: String;
```

```
begin
```

```
    // Show the contents of Readme.txt in a message box
```

```
    ExtractTemporaryFile('Readme.txt');
```

```
    if LoadStringFromFile(ExpandConstant('{tmp}\Readme.txt'), S) then
```

```
    begin
```

```
        MsgBox(S, mbInformation, MB_OK);
```

```
    end;
```

```
    Result := True;
```

```
end;
```

## **Support function: GetPreviousData**

### **Prototype:**

```
function GetPreviousData(const ValueName, DefaultValueData: String): String;
```

### **Description:**

Gets a value that was previously stored using SetPreviousData.

## Support function: 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.

## **Support function: Terminated**

### **Prototype:**

```
function Terminated: Boolean;
```

### **Description:**

Returns True if Setup or Uninstall is terminating, False otherwise.

## Support function: 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)
```

## Support function: 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 of the exception
  MsgBox('You will not see this.', mbInformation, MB_OK);
end;
```

## Support function: `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 zero" exception
    I := I div 0;
  except
    // Catch the exception, deal with it, and continue
    MsgBox('We caught this exception: ' + AddPeriod(GetExceptionMessage),
      mbError, MB_OK);
  end;
end;
```

## Support function: 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 `/LOG` switch) the message will be 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 zero" exception
    I := I div 0;
  except
    // Catch the exception, show it, and continue
    ShowExceptionMessage;
  end;
end;
```

## **Support function: IsAdminLoggedOn**

### **Prototype:**

```
function IsAdminLoggedOn: Boolean;
```

### **Description:**

Returns True if an administrator is logged onto the system. Always returns True on Windows 95/98/ME.

## **Support function: IsPowerUserLoggedOn**

### **Prototype:**

```
function IsPowerUserLoggedOn: Boolean;
```

### **Description:**

Returns True if a Power User is logged onto the system. Always returns True on Windows 95/98/ME.

## **Support function: UsingWinNT**

### **Prototype:**

```
function UsingWinNT: Boolean;
```

### **Description:**

Returns True if system is running a version of Windows that has an NT-based kernel (e.g. Windows NT 4.0, 2000, XP, 2003).

## Support function: **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".

## **Support function: 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".

## **Support function: InstallOnThisVersion**

### **Prototype:**

```
function InstallOnThisVersion(const MinVersion, OnlyBelowVersion: String):  
Integer;
```

### **Description:**

Returns `irlInstall` if an entry with the specified `MinVersion` and `OnlyBelowVersion` parameters should be installed.

## **Support function: GetEnv**

### **Prototype:**

```
function GetEnv(const EnvVar: String): String;
```

### **Description:**

Gets the value of the specified environment variable.

## **Support function: GetUserNameString**

### **Prototype:**

```
function GetUserNameString: String;
```

### **Description:**

Retrieves the name of the user currently logged onto the system.

## **Support function: GetComputerNameString**

### **Prototype:**

```
function GetComputerNameString: String;
```

### **Description:**

Retrieves the computer name of the current system.

## Support function: 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:

See [http://msdn.microsoft.com/library/en-us/intl/nls\\_238z.asp](http://msdn.microsoft.com/library/en-us/intl/nls_238z.asp) for a list of possible language identifiers.

### Example:

```
begin
  if GetUILanguage = $0409 then
    begin
      // UI language is English (United States)
    end;

    // You can use "and $3FF" to extract the primary language identifier
    if GetUILanguage and $3FF = $09 then
      begin
        // Matches any variant of English
      end;
    end;
end;
```

## **Support function: 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.

## **Support function: 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.

## **Support function: 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.

## **Support function: 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.

## **Support function: SendMessage**

### **Prototype:**

```
function SendMessage(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.

## **Support function: 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.

## **Support function: 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.

## **Support function: 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.

## **Support function: SendBroadcastNotifyMessage**

### **Prototype:**

```
function SendBroadcastNotifyMessage(const Msg, WParam, LParam: Longint):  
Boolean;
```

### **Description:**

*not yet available*

## **Support function: CreateMutex**

### **Prototype:**

```
procedure CreateMutex(const Name: String);
```

### **Description:**

Creates a mutex with the specified name.

## **Support function: CheckForMutexes**

### **Prototype:**

```
function CheckForMutexes(Mutexes: String): Boolean;
```

### **Description:**

Returns True if any of the mutexes in the comma-separated Mutexes string exist.

## Support function: MakePendingFileRenameOperationsChecksum

### Prototype:

```
procedure MakePendingFileRenameOperationsChecksum: String;
```

### Description:

Calculates a checksum of the current PendingFileRenameOperations registry value (on NT 4+ platforms) or of the current WININIT.INI file (on non-NT platforms). 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 := MakePendingFileRenameOperationsChecksum;
  // ...run a program...
  ChecksumAfter := MakePendingFileRenameOperationsChecksum;
  if ChecksumAfter <> ChecksumBefore then
    // PendingFileRenameOperations or WININIT.INI changed
end;
```

## Support function: 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 stdcall uninstallonly';  
  
...  
  
begin  
    // Call DllFunc  
    DllFunc;  
  
    // Unload the DLL  
    UnloadDLL(ExpandConstant('{app}\MyDll.dll'));  
  
    // Now we can delete the DLL  
    DeleteFile(ExpandConstant('{app}\MyDll.dll'));  
end;
```

## Support function: 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.

### Example:

```
function MessageBox(hWnd: Integer; lpText, lpCaption: String; uType: Cardinal): Integer; external 'MessageBoxA@user32.dll stdcall';
```

```
...
```

```
begin
```

```
  if MessageBox(-1, '', '', -1) = 0 then
```

```
    MsgBox(SysErrorMessage(DLLGetLastError), mbError, mb_Ok);
```

## **Support function: Chr**

### **Prototype:**

```
function Chr(B: Byte): Char;
```

### **Description:**

Returns the character with the specified ordinal value.

## **Support function: Ord**

### **Prototype:**

```
function Ord(C: Char): Byte;
```

### **Description:**

Returns the ordinal value of the specified character.

## Support function: Copy

### Prototype:

```
function Copy(S: String; Indx, Count: Integer): String;
```

### Description:

Returns a string containing Count characters starting with 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, the only the characters from S[Index] to the end of S are returned.

## **Support function: Length**

### **Prototype:**

```
function Length(s: String): Longint;
```

### **Description:**

Returns the length of the specified string.

## **Support function: Lowercase**

### **Prototype:**

```
function Lowercase(s: string): String;
```

### **Description:**

Returns a string with the same text as the string passed in S, but with all letters converted to lowercase.

## **Support function: StringOfChar**

### **Prototype:**

```
function StringOfChar(c: Char; I : Longint): String;
```

### **Description:**

Returns a string of length I with all characters set to character C.

## **Support function: Uppercase**

### **Prototype:**

```
function Uppercase(s: string): String;
```

### **Description:**

Returns a string containing the same text as S, but with all letters converted to uppercase.

## **Support function: Delete**

### **Prototype:**

```
procedure Delete(var S: String; Indx, 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.

## **Support function: Insert**

### **Prototype:**

```
procedure Insert(Source: String; var Dest: String; Indx: Integer);
```

### **Description:**

Merges Source into S at the position S[index].

## **Support function: StringChange**

### **Prototype:**

```
procedure StringChange(var S: String; const FromStr, ToStr: String);
```

### **Description:**

Change all occurrences in S of FromStr to ToStr.

## **Support function: 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.

## **Support function: 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.

## **Support function: RemoveQuotes**

### **Prototype:**

```
function RemoveQuotes(const S: String): String;
```

### **Description:**

Opposite of AddQuotes; removes any quotes around the string.

## **Support function: ConvertPercentStr**

### **Prototype:**

```
function ConvertPercentStr(var S: String): Boolean;
```

### **Description:**

Expands all %-encoded characters in the string (see RFC 2396). Returns True if all were successfully expanded.

## **Support function: 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.

## **Support function: 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.

## **Support function: Format1**

### **Prototype:**

```
function Format1(const Format, S1: String): String;
```

### **Description:**

Returns the Format string with the first %s in the Format string replaced by the S1 string.

## **Support function: Format2**

### **Prototype:**

```
function Format2(const Format, S1, S2: String): String;
```

### **Description:**

Returns the Format string with the first %s in the Format string replaced by the S1 string and the second %s replaced by the S2 string.

## Support function: Format3

### Prototype:

```
function Format3(const Format, S1, S2, S3: String): String;
```

### Description:

Returns the Format string with the first %s in the Format string replaced by the S1 string, the second %s replaced by the S2 string, etc.

## **Support function: Format4**

### **Prototype:**

```
function Format4(const Format, S1, S2, S3, S4: String): String;
```

### **Description:**

Returns the Format string with the first %s in the Format string replaced by the S1 string, the second %s replaced by the S2 string, etc.

## **Support function: Trim**

### **Prototype:**

```
function Trim(const S: string): String;
```

### **Description:**

Trims leading and trailing spaces and control characters from the given string S.

## **Support function: TrimLeft**

### **Prototype:**

```
function TrimLeft(const S: string): String;
```

### **Description:**

Trims leading spaces and control characters from the given string S.

## **Support function: TrimRight**

### **Prototype:**

```
function TrimRight(const S: string): String;
```

### **Description:**

Trims trailing spaces and control characters from the given string S.

## Support function: 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.

## **Support function: StrToInt**

### **Prototype:**

```
function StrToInt(s: string): Longint;
```

### **Description:**

The StrToInt function converts the string passed in S into a number.

### **Remarks:**

Use of StrToIntDef instead of StrToInt is recommended.

## **Support function: IntToStr**

### **Prototype:**

```
function IntToStr(i: Longint): String;
```

### **Description:**

The IntToStr function converts an integer into a string containing the decimal representation of that number.

## Support function: 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 increment I by 1.
    // Increment by CharLength() in case Result[I] is a double-byte
    character.
    I := I + CharLength(Result, I);
  end;
end;

...

begin
  // Show path of Common Files with backslashes changed to forward slashes
  MsgBox(BackslashToSlash(ExpandConstant('{cf}')), mbInformation, MB_OK);
end;
```

## Support function: 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.

## **Support function: 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.

## Support function: 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 [RemoveBackslashUnlessRoot](#) instead when working with file system paths.

## **Support function: 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.

## **Support function: 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.

## **Support function: 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.

## Support function: 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.

## Support function: 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 leftmost 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.

## Support function: 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.

## Support function: 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.

### Example:

```
var
  S: String;
begin
  S := ExtractRelativePath('c:\windows\system32\', 'c:\autoexec.bat');
  // S = ..\..\autoexec.bat
end;
```

## **Support function: 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.

## Support function: 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.

## Support function: 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).
- dddddd Displays the date using the system's short date format.
- ddddddd Displays the date using the system's long date format.
- m Displays the month as a number without a leading zero (1-12).  
If the m specifier immediately follows an h or hh specifier, the minute rather than the month is displayed.
- mm Displays the month as a number with a leading zero (01-12).  
If the mm specifier immediately follows an h or hh specifier, the minute rather than the month is displayed.
- mmm Displays the month as an abbreviation (Jan-Dec).
- mmmm Displays the month as a full name (January-December).
- yy Displays the year as a two-digit number (00-99).
- yyyy Displays the year as a four-digit number (0000-9999).
- h Displays the hour without a leading zero (0-23).
- hh Displays the hour with a leading zero (00-23).
- n Displays the minute without a leading zero (0-59).
- nn Displays the minute with a leading zero (00-59).
- s Displays the second without a leading zero (0-59).
- ss Displays the second with a leading zero (00-59).
- t Displays the time using the system's short time format.
- tt Displays the time using the system's long time format.
- am/pm Uses the 12-hour clock for the preceding h or hh specifier.  
Displays 'am' for any hour before noon, and 'pm' for any hour after noon.  
The am/pm specifier can use lower, upper, or mixed case, and the result is displayed accordingly.
- a/p Uses the 12-hour clock for the preceding h or hh specifier.  
Displays 'a' for any hour before noon, and 'p' for any hour after noon.  
The a/p specifier can use lower, upper, or mixed case, and the result is displayed accordingly.
- / Displays the date separator character given by the DateSeparator parameter.  
If DateSeparator is set to #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.

### Example:

```
GetDateTimeString('dddddd', #0, #0);  
GetDateTimeString('dddddd tt', #0, #0);  
GetDateTimeString('dd/mm/yyyy hh:nn:ss', '-', ':');
```

## **Support function: SetLength**

### **Prototype:**

```
procedure SetLength(var S: String; L: Longint);
```

### **Description:**

Sets the length of a string.

## **Support function: CharToOemBuff**

### **Prototype:**

```
procedure CharToOemBuff(var S: String);
```

### **Description:**

Translates an ANSI string to a string with characters from the OEM-defined character set.

## **Support function: OemToCharBuff**

### **Prototype:**

```
procedure OemToCharBuff(var S: String);
```

### **Description:**

Translates a string with characters from the OEM-defined character set into an ANSI string.

## Support function: GetMD5OfString

### Prototype:

```
function GetMD5OfString(const S: String): String;
```

### Description:

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

### Example:

```
var
  MD5: String;
begin
  MD5 := GetMD5OfString('Test');
  // MD5 = '0cbc6611f5540bd0809a388dc95a615b'
end;
```

## **Support function: SysErrorMessage**

### **Prototype:**

```
function SysErrorMessage(ErrorCode: Integer): String;
```

### **Description:**

Returns a localized error message string that corresponds to the given operating system error code.

## **Support function: GetArrayLength**

### **Prototype:**

```
function GetArrayLength(var Arr: Array): Longint;
```

### **Description:**

Gets the length of an array.

## **Support function: 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.

## **Support function: 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.

## **Support function: FileExists**

### **Prototype:**

```
function FileExists(const Name: String): Boolean;
```

### **Description:**

Returns True if the specified file exists.

## Support function: 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.

## **Support function: 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.

## Support function: 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 Files drive
  Path := ExpandConstant('{pf}');
  if GetSpaceOnDisk(Path, True, FreeMB, TotalMB) then
  begin
    MsgBox('There are ' + IntToStr(FreeMB) + ' megabytes free on ' +
      Path, mbInformation, MB_OK);
  end
  else begin
    // the function failed
  end;
end;
```

## **Support function: 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.

## Support function: 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;           // name of the found file (no path)
  Attributes: LongWord;  // file attributes
  SizeHigh: LongWord;    // size of the file, upper 32 bits
  SizeLow: LongWord;     // size of the file, lower 32 bits
  FindHandle: THandle;   // used internally
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
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 FindNext to enumerate the rest of the files, and then FindClose.

### 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) then begin
    try
      repeat
        // Don't count directories
        if FindRec.Attributes and FILE_ATTRIBUTE_DIRECTORY = 0 then
          FilesFound := FilesFound + 1;
      until not FindNext(FindRec);
    finally
      FindClose(FindRec);
    end;
  end;
  MsgBox(IntToStr(FilesFound) + ' files found in the System directory.',
    mbInformation, MB_OK);
```

end;

## **Support function: FindNext**

### **Prototype:**

```
function FindNext(var FindRec: TFindRec): Boolean;
```

### **Description:**

Retrieves information about the next matching file after a call to FindFirst. Returns True if successful.

### **Example:**

For an example, see the documentation for FindFirst.

## **Support function: 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 FindFirst returns True.

### **Example:**

For an example, see the documentation for FindFirst.

## **Support function: GetCurrentDir**

### **Prototype:**

```
function GetCurrentDir: String;
```

### **Description:**

Returns a string containing the name of the current directory.

## Support function: **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.

## **Support function: 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.

## **Support function: 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.

## Support function: 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.

## Support function: 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, mbInformation, MB_OK);
    end
  else
    begin
      // handle failure
    end;
end;
```

## Support function: 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 ShlObj.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, mbInformation, MB_OK);  
  end  
  else  
  begin  
    // handle failure  
  end;  
end;
```

## **Support function: 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.

## **Support function: 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.

## **Support function: GetVersionNumbers**

### **Prototype:**

```
function GetVersionNumbers(const Filename: String; var VersionMS, VersionLS: Cardinal): Boolean;
```

### **Description:**

Gets the file version numbers of the specified file.

## **Support function: GetVersionNumbersString**

### **Prototype:**

```
function GetVersionNumbersString(const Filename: String; var Version:  
String): Boolean;
```

### **Description:**

Gets the file version numbers of the specified file, as a string.

## **Support function: 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).

## **Support function: GetMD5OfFile**

### **Prototype:**

```
function GetMD5OfFile(const Filename: String): String;
```

### **Description:**

Gets the MD5 sum of the specified file, as a string. An exception will be raised upon failure.

## Support function: 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. 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 [ShellExec](#) 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.

### Example:

```
var
  ResultCode: Integer;
begin
  // Launch Notepad and wait for it to terminate
  if Exec(ExpandConstant('{win}\notepad.exe'), '', '', SW_SHOW,
    ewWaitUntilTerminated, ResultCode) then
  begin
    // handle success if necessary; ResultCode contains the exit code
  end
  else begin
    // handle failure if necessary; ResultCode contains the error code
  end;
end;
```

## Support function: 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. The filename can be an executable file, a document file, or a folder. 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);
```

### Example:

```
var
  ErrorCode: Integer;
begin
  if not ShellExec('open', ExpandConstant('{app}\filename.rtf'),
    '', '', SW_SHOW, ewNoWait, ErrorCode) then
    begin
      // handle failure if necessary
    end;
end;
```

## **Support function: 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.

## **Support function: 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.

## Support function: 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.

## **Support function: 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.

## **Support function: 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.

## **Support function: LoadStringFromFile**

### **Prototype:**

```
function LoadStringFromFile(const FileName: String; var S: String): Boolean;
```

### **Description:**

Loads the specified binary or text file into the specified string. Returns True if successful, False otherwise.

## **Support function: 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.

## Support function: SaveStringToFile

### Prototype:

```
function SaveStringToFile(const FileName, S: String; 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);
```

## **Support function: SaveStringsToFile**

### **Prototype:**

```
function SaveStringsToFile(const FileName: String; const S: TArrayOfString;  
const Append: Boolean): Boolean;
```

### **Description:**

Saves the specified string array 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.

## **Support function: 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.

## **Support function: 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.

## **Support function: 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.

## Support function: 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 inside it  
  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;
```

## Support function: 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.

### Example:

```
CreateShellLink(
  ExpandConstant('{app}\config\Open licence database.lnk'),
  'Opens the licence database in SQLite',
  ExpandConstant('{app}\config\sqlite.exe'),
  ExpandConstant('{app}\config\licences.db'),
  ExpandConstant('{app}\config'),
  '',
  0,
  SW_SHOWNORMAL);
```

## **Support function: RegisterServer**

### **Prototype:**

```
procedure RegisterServer(const Filename: String; const FailCriticalErrors: Boolean);
```

### **Description:**

Registers the OLE server (a.k.a. ActiveX control) with the specified filename. If FailCriticalErrors is True, the system is allowed to display error messages. Throws an exception if not successful.

## **Support function: UnregisterServer**

### **Prototype:**

```
function UnregisterServer(const Filename: String; const FailCriticalErrors:  
Boolean): Boolean;
```

### **Description:**

Unregisters the OLE server (a.k.a. ActiveX control) with the specified filename. If FailCriticalErrors is True, the system is allowed to display error messages. Returns True if successful, False otherwise.

## **Support function: RegisterTypeLibrary**

### **Prototype:**

```
procedure RegisterTypeLibrary(const Filename: String);
```

### **Description:**

Registers the type library with the specified filename. Throws an exception if not successful.

## **Support function: UnregisterTypeLibrary**

### **Prototype:**

```
function UnregisterTypeLibrary(const Filename: String): Boolean
```

### **Description:**

Unregisters the type library with the specified filename. Returns True if successful, False otherwise.

## Support function: IncrementSharedCount

### Prototype:

```
procedure IncrementSharedCount(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\CurrentVersion\SharedDLLs

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.

## Support function: DecrementSharedCount

### Prototype:

```
function DecrementSharedCount(const Filename: String): Boolean;
```

### Description:

Decrements the reference count for the specified file in the following registry key:

HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\SharedDLLs

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.

## **Support function: 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.

## **Support function: UnregisterFont**

### **Prototype:**

```
procedure UnregisterFont(const FontName, FontFilename: String);
```

### **Description:**

Unregisters the font with the specified face and filename.

## **Support function: 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.

## Support function: RegKeyExists

### Prototype:

```
function RegKeyExists(const RootKey: Integer; const SubKeyName: String):  
Boolean;
```

### Description:

Returns True if the specified registry key exists.

### Example:

```
begin  
  if RegKeyExists(HKEY_CURRENT_USER, 'Software\Jordan Russell\Inno Setup')  
then  
  begin  
    // The key exists  
  end;  
end;
```

## Support function: RegValueExists

### Prototype:

```
function RegValueExists(const RootKey: Integer; const SubKeyName, ValueName: String): Boolean;
```

### Description:

Returns True if the specified registry key and value exist.

### Example:

```
begin
  if RegValueExists(HKEY_CURRENT_USER, 'Console', 'WindowSize') then
    begin
      // The value exists
    end;
end;
```

## Support function: 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.

### Example:

```
var  
  Names: TArrayOfString;  
  I: Integer;  
  S: String;  
begin  
  if RegGetSubkeyNames(HKEY_CURRENT_USER, 'Control Panel', Names) then  
  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, mbInformation, MB_OK);  
  end else  
  begin  
    // add any code to handle failure here  
  end;  
end;
```

## Support function: 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.

### Example:

```
var  
  Names: TArrayOfString;  
  I: Integer;  
  S: String;  
begin  
  if RegGetValueNames(HKEY_CURRENT_USER, 'Control Panel\Mouse', Names) then  
  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, mbInformation, MB_OK);  
  end else  
  begin  
    // add any code to handle failure here  
  end;  
end;
```

## Support function: 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 Panel\International',  
        'sCountry', Country) then  
        begin  
            // Successfully read the value  
            MsgBox('Your country: ' + Country, mbInformation, MB_OK);  
        end;  
end;
```

## Support function: 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).

## Support function: 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(HistoryBufferSize),  
                mbInformation, MB_OK);  
        end;  
    end;  
end;
```

## Support function: RegQueryBinaryValue

### Prototype:

```
function RegQueryBinaryValue(const RootKey: Integer; const SubKeyName,  
ValueName: String; var ResultStr: String): 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.

## Support function: 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.

### Example:

```
begin  
  RegWriteStringValue(HKEY_CURRENT_USER, 'Software\My Company\My Program',  
    'UserName', ExpandConstant('{sysuserinfoname}'));  
end;
```

## Support function: 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 Company\My Program',  
        'UserName', '%UserName%');  
end;
```

## Support function: 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, 'Software\My Company\My  
Program',  
    'MultiStringTest', 'String1' + #0 + 'String2' + #0 + 'String3');  
end;
```

## Support function: 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  
    RegWriteDWordValue(HKEY_CURRENT_USER, 'Software\My Company\My Program',  
        'ShowToolbar', 1);  
end;
```

## Support function: RegWriteBinaryValue

### Prototype:

```
function RegWriteBinaryValue(const RootKey: Integer; const SubKeyName,  
ValueName, Data: String): Boolean;
```

### Description:

Writes the specified REG\_BINARY-type registry value. Returns True if successful, False otherwise.

### Example:

```
begin  
  RegWriteBinaryValue(HKEY_CURRENT_USER, 'Software\My Company\My Program',  
    'BinaryTest', 'Whatever' + #1#2#3#4);  
end;
```

## **Support function: 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.

## Support function: 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.

## **Support function: RegDeleteValue**

### **Prototype:**

```
function RegDeleteValue(const RootKey: Integer; const SubKeyName, ValueName:  
String): Boolean;
```

### **Description:**

Deletes the specified value. Returns True if successful, False otherwise.

## **Support function: IniKeyExists**

### **Prototype:**

```
function IniKeyExists(const Section, Key, Filename: String): Boolean;
```

### **Description:**

Returns True if the specified INI key exists.

## **Support function: IsIniSectionEmpty**

### **Prototype:**

```
function IsIniSectionEmpty(const Section, Filename: String): Boolean;
```

### **Description:**

Returns True if the specified INI section is empty.

## **Support function: GetIniBool**

### **Prototype:**

```
function GetIniBool(const Section, Key: String; const Default: Boolean; const  
Filename: String): Boolean
```

### **Description:**

Reads a Boolean from an INI file.

## **Support function: 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.

## **Support function: GetIniString**

### **Prototype:**

```
function GetIniString(const Section, Key, Default, Filename: String): String;
```

### **Description:**

Reads a String from an INI file.

## **Support function: SetIniBool**

### **Prototype:**

```
function SetIniBool(const Section, Key: String; const Value: Boolean; const  
Filename: String): Boolean;
```

### **Description:**

Writes a Boolean to an INI file.

## **Support function: SetIniInt**

### **Prototype:**

```
function SetIniInt(const Section, Key: String; const Value: Longint; const  
Filename: String): Boolean;
```

### **Description:**

Writes a Longint to an INI file.

## **Support function: SetIniString**

### **Prototype:**

```
function SetIniString(const Section, Key, Value, Filename: String): Boolean;
```

### **Description:**

Writes a string to an INI file.

## **Support function: DeleteIniSection**

### **Prototype:**

```
procedure DeleteIniSection(const Section, Filename: String);
```

### **Description:**

Deletes the specified section from an INI file.

## **Support function: DeleteIniEntry**

### **Prototype:**

```
procedure DeleteIniEntry(const Section, Key, Filename: String);
```

### **Description:**

Deletes the specified key from an INI file.

## Support function: 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 you work, then click  
Next.');
```

```
// 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];
```

## Support function: 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 below, then click  
Next.',  
  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];
```

## Support function: 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 be  
stored?',  
    'Personal data files will be stored in the following folder.'#13#10#13#10 +  
    'To continue, click Next. If you would like to select a different folder,  
click Browse.',  
    False, 'New Folder');  
  
// Add item (with an empty caption)  
Page.Add('');  
  
// Set initial value (optional)  
Page.Values[0] := ExpandConstant('{userappdata}\My Company\My Program');  
  
...  
  
// Read value into variable  
DataDir := Page.Values[0];
```

## Support function: 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:',           // caption  
  'Executable files|*.exe|All files|*.*',     // filters  
  '.exe');                                     // default extension
```

```
// Set initial value (optional)  
Page.Values[0] := ExpandConstant('{win}\notepad.exe');
```

```
...  
  
// Read value into variable  
NotepadLocation := Page.Values[0];
```

## Support function: 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 information before  
continuing.',  
    'Blah blah blah.');
```

## Support function: CreateOutputMsgMemoPage

### Prototype:

```
function CreateOutputMsgMemoPage(const AfterID: Integer; const ACaption,  
ADescription, ASubCaption, AMsg: String): 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 information before  
continuing.',  
    'When you are ready to continue with Setup, click Next.',  
    'Blah blah blah.');
```

## Support function: 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.

## Support function: 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 TWizardPage instance returned by this function).

### Example:

See *CodeClasses.iss* for an example.

## Support function: 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 TSetupForm instance returned by this function).

### Remarks:

You should call this function instead of creating TForm or TSetupForm 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.

## Support function: PageFromID

### Prototype:

```
function PageFromID(const ID: Integer): TWizardPage;
```

### Description:

Given a page ID, returns a TWizardPage 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;
```

## **Support function: 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.

## **Support function: 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.

## Support function: 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) = IDYES then
  begin
    // user clicked Yes
    end;

  // Ask the user a Yes/No question, defaulting to No
  if MsgBox('Are you sure?', mbConfirmation, MB_YESNO or MB_DEFBUTTON2) =
  IDYES then
  begin
    // user clicked Yes
    end;
end;
```

## Support function: 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 [Setup Command Line Parameters](#)), `Default` is returned. Otherwise, `SuppressibleMsgBox` acts the same as the regular [MsgBox](#).

## Support function: 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 a 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('Select File', Filename, '',  
    'Text Documents (*.txt)|*.txt|All Files|*.*', 'txt') then  
  begin  
    // Successful; user clicked OK  
    // Filename contains the selected filename  
  end;  
end;
```

## Support function: 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.

## **Support function: 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.

## Support function: CreateOleObject

### Prototype:

```
function CreateOleObject(const ClassName: string): Variant;
```

### Description:

See the [Using COM Automation objects](#) topic.

## Support function: GetActiveOleObject

### Prototype:

```
function GetActiveOleObject(const ClassName: string): Variant;
```

### Description:

See the [Using COM Automation objects](#) topic.

## **Support function: CoFreeUnusedLibraries**

### **Prototype:**

procedure CoFreeUnusedLibraries;

### **Description:**

See the [Using COM Automation objects](#) topic.

## Support function: 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 by the /LOG command line parameter. See the [Setup Command Line Parameters](#) topic.

## **Support function: Sleep**

### **Prototype:**

```
procedure Sleep(const Milliseconds: LongInt);
```

### **Description:**

Suspends the execution of Setup or Uninstall for a specified interval.

## **Support function: Random**

### **Prototype:**

```
function Random(const Range: Integer): Integer;
```

### **Description:**

Returns a random number within the range  $0 \leq X < \text{Range}$ .

## **Support function: Beep**

### **Prototype:**

```
procedure Beep;
```

### **Description:**

Beeps.

### **Example:**

```
Beep; //Beeps
```

## **Support function: BringToFrontAndRestore**

### **Prototype:**

```
procedure BringToFrontAndRestore;
```

### **Description:**

Makes sure that Setup or Uninstall is visible and the foreground window.

## Support function: 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 [Using DLLs](#) topic.

## Support function: 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 if 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:

This function is deprecated. See the [Using DLLs](#) topic.

## Support function: FreeDLL

### Prototype:

```
function FreeDLL(const DLLHandle: Longint): Boolean;
```

### Description:

Unloads a DLL specified using the DLL handle returned by LoadDLL.

### Remarks:

This function is deprecated. See the [Using DLLs](#) topic.

## Support function: 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:

This function is deprecated. See the [Using DLLs](#) topic.

## Support function: 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 CallDllProc.

### Remarks:

This function is deprecated. See the [Using DLLs](#) topic.



## Pascal Scripting: Support Classes Reference

Below is the list of support classes that can be used from within the Pascal script. There are also two support objects available: `MainForm` of type `TMainForm` and `WizardForm` of type `TWizardForm` and one special constant: `crHand` of type `TControl.Cursor`. Note: `MainForm` is only visible if WindowVisible is set to `yes`.

Note: you may find it useful to also refer to the Delphi Visual Component Library (VCL) Help files by Borland, since the classes below are mostly simple wrappers around the VCL classes Inno Setup uses internally. See <http://info.borland.com/techpubs/delphi/> and <ftp://ftp.borland.com/pub/delphi/techpubs/delphi3/d3cs.zip>.

```

TObject = class
    constructor Create;
    procedure Free;
end;

TPersistent = class(TObject)
    procedure Assign(Source: TPersistent);
end;

TComponent = class(TPersistent)
    function FindComponent(AName: string): TComponent;
    constructor Create(AOwner: TComponent);

    property Owner: TComponent; read write;
    procedure DESTROYCOMPONENTS;
    procedure DESTROYING;
    procedure FREENOTIFICATION(ACOMPONENT: TCOMPONENT);
    procedure INSERTCOMPONENT(ACOMPONENT: TCOMPONENT);
    procedure REMOVECOMPONENT(ACOMPONENT: TCOMPONENT);
    property COMPONENTS[Index: INTEGER]: TCOMPONENT; 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(TPersistent)
    function Add(S: string): Integer;
    procedure Append(S: string);
    procedure AddStrings(Strings: TStrings);
    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: TObject);

TStringList = class(TStrings)
    function FIND(S: STRING; var INDEX: INTEGER): BOOLEAN;
    procedure SORT;
    property DUPLICATES: TDUPLICATES; read write;
    property SORTED: BOOLEAN; read write;
    property ONCHANGE: TNOTIFYEVENT; read write;
    property ONCHANGING: TNOTIFYEVENT; read write;
end;

TStream = class(TObject)

```

```

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;
property POSITION: LONGINT; read write;
property SIZE: LONGINT; read write;
end;

THandleStream = class (TStream)
  constructor CREATE (AHANDLE:INTEGER);
  property HANDLE: INTEGER; read;
end;

TFileStream = class (THandleStream)
  constructor CREATE (FILENAME:STRING;MODE:WORD);
end;

TGraphicsObject = class (TPersistent)
  property ONCHANGE: TNOTIFYEVENT; read write;
end;

TFontStyle = (fsBold, fsItalic, fsUnderline, fsStrikeOut);

TFontStyles = set of TFontStyle;

TFont = class (TGraphicsObject)
  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: TFontStyles; read write;
end;

TCanvas = class (TPersistent)
  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: TGraphic);
  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: TBrush; read;

```

```

    property CopyMode: Byte; read write;
    property Font: TFont; read;
    property Pen: TPen; 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(TGraphicsObject)
    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(TPersistent)
    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: TNotifyEvent; read write;
end;

TBitmap = class(TGraphic)
    procedure LoadFromStream(Stream: TStream);
    procedure SaveToStream( Stream: TStream);
    property Canvas: TCanvas; read write;
    property Handle: HBITMAP; read write;
end;

TAlign = (alNone, alTop, alBottom, alLeft, alRight, alClient);

TControl = class(TComponent)
    constructor Create(AOwner: TComponent);
    procedure BringToFront;
    procedure Hide;
    procedure Invalidate;
    procedure refresh;
    procedure Repaint;
    procedure SendToBack;
    procedure Show;

```

```

procedure Update;
procedure SetBounds(x,y,w,h: 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: TAlign; 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: TWinControl; 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]: TCONTROL; read;
  property CONTROLCOUNT: INTEGER; read;
end;

TGraphicControl = class(TControl)
end;

TCustomControl = class(TWinControl)
end;

TScrollBarKind = (sbHorizontal, sbVertical);

TScrollBarInc = SmallInt;

TControlScrollBar = class('TPersistent')
  property KIND: TSCROLLBARKIND; read;
  property SCROLLPOS: INTEGER; read;
  property MARGIN: WORD; read write;
  property INCREMENT: TSCROLLBARINC; read write;
  property RANGE: INTEGER; read write;
  property POSITION: INTEGER; read write;
  property TRACKING: BOOLEAN; read write;
  property VISIBLE: BOOLEAN; read write;
end;

TScrollingWinControl = class(TWinControl)
  procedure SCROLLINVIEW(ACONTROL:TCONTROL);
  property HORIZSCROLLBAR: TCONTROLSCROLLBAR; read write;
  property VERTSCROLLBAR: TCONTROLSCROLLBAR; read write;
end;

TFormBorderStyle = (bsNone, bsSingle, bsSizeable, bsDialog, bsToolWindow,

```

```

bsSizeToolWin);

TBorderIcon = (biSystemMenu, biMinimize, biMaximize, biHelp);

TBorderIcons = set of TBorderIcon;

TPosition = (poDesigned, poDefault, poDefaultPosOnly, poDefaultSizeOnly,
poScreenCenter, poDesktopCenter, poMainFormCenter, poOwnerFormCenter);

TCloseAction = (caNone, caHide, caFree, caMinimize);

TCloseEvent = procedure(Sender: TObject; var Action: TCloseAction);

TCloseQueryEvent = procedure(Sender: TObject; var CanClose: Boolean);

TShiftState = (ssShift, ssAlt, ssCtrl, ssLeft, ssRight, ssMiddle,
ssDouble);

TShiftState = set of TShiftState;

TKeyEvent = procedure (Sender: TObject; var Key: Word; Shift: TShiftState);

TKeyPressEvent = procedure(Sender: TObject; var Key: Char);

TForm = class(TScrollingWinControl)
    constructor CREATENEW(AOWNER:TCOMPONENT; Dummy: Longint);
    procedure CLOSE;
    procedure HIDE;
    procedure SHOW;
    function SHOWMODAL:INTEGER;
    procedure RELEASE;
    property ACTIVE: BOOLEAN; read;
    property ACTIVECONTROL: TWINCONTROL; read write;
    property BORDERICONS: TBorderIcons; read write;
    property BORDERSTYLE: TFORMBORDERSTYLE; 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: TPOSITION; read write;
    property ONACTIVATE: TNOTIFYEVENT; read write;
    property ONCLICK: TNOTIFYEVENT; read write;
    property ONDBLCLICK: TNOTIFYEVENT; read write;
    property ONCLOSE: TCLOSEEVENT; read write;
    property ONCLOSEQUERY: TCLOSEQUERYEVENT; read write;
    property ONCREATE: TNOTIFYEVENT; read write;
    property ONDESTROY: TNOTIFYEVENT; read write;
    property ONDEACTIVATE: TNOTIFYEVENT; read write;
    property ONHIDE: TNOTIFYEVENT; read write;
    property ONKEYDOWN: TKEYEVENT; read write;
    property ONKEYPRESS: TKEYPRESSEVENT; read write;
    property ONKEYUP: TKEYEVENT; read write;
    property ONRESIZE: TNOTIFYEVENT; read write;
    property ONSHOW: TNOTIFYEVENT; read write;
end;

```

```

TCustomLabel = class(TGraphicControl)
end;

TAlignment = (taLeftJustify, taRightJustify, taCenter);

TLabel = class(TCustomLabel)
  property ALIGNMENT: TAlignment; read write;
  property AUTOSIZE: Boolean; read write;
  property CAPTION: String; read write;
  property COLOR: TColor; read write;
  property FOCUSCONTROL: TWinControl; read write;
  property FONT: TFont; read write;
  property WORDWRAP: Boolean; read write;
  property ONCLICK: TNotifyEvent; read write;
  property ONDBLCLICK: TNotifyEvent; 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(TCustomEdit)
  property AUTOSELECT: Boolean; read write;
  property AUTOSIZE: Boolean; read write;
  property BORDERSTYLE: TBorderStyle; read write;
  property CHARCASE: TEditCharCase; read write;
  property COLOR: TColor; read write;
  property FONT: TFont; 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: TNotifyEvent; read write;
  property ONCLICK: TNotifyEvent; read write;
  property ONDBLCLICK: TNotifyEvent; read write;
  property ONKEYDOWN: TKeyEvent; read write;
  property ONKEYPRESS: TKeyPressEvent; read write;
  property ONKEYUP: TKeyEvent; read write;
end;

TCustomMemo = class(TCustomEdit)
  property LINES: TSTRINGS; read write;
end;

```

```
TScrollStyle = (ssNone, ssHorizontal, ssVertical, ssBoth);
```

```
TMemo = class(TMemo)
  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: TNotifyEvent; read write;
  property ONCLICK: TNotifyEvent; read write;
  property ONDBLCLICK: TNotifyEvent; read write;
  property ONKEYDOWN: TKeyEvent; read write;
  property ONKEYPRESS: TKeyPressEvent; read write;
  property ONKEYUP: TKeyEvent; read write;
end;
```

```
TCustomComboBox = class(TWinControl)
  property DROPPEDDOWN: BOOLEAN; read write;
  property ITEMS: TSTRINGS; read write;
  property ITEMINDEX: INTEGER; read write;
end;
```

```
TComboBoxStyle = (csDropDown, csSimple, csDropDownList, csOwnerDrawFixed,
csOwnerDrawVariable);
```

```
TComboBox = class(TCustomComboBox)
  property STYLE: TComboBoxStyle; 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: TNotifyEvent; read write;
  property ONCLICK: TNotifyEvent; read write;
  property ONDBLCLICK: TNotifyEvent; read write;
  property ONDROPDOWN: TNotifyEvent; read write;
  property ONKEYDOWN: TKeyEvent; read write;
  property ONKEYPRESS: TKeyPressEvent; read write;
  property ONKEYUP: TKeyEvent; read write;
end;
```

```
TButtonControl = class(TWinControl)
end;
```

```
TButton = class(TButtonControl)
  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: TNotifyEvent; read write;
end;

TCustomCheckBox = class(TButtonControl)
end;

TCheckBoxState = (cbUnchecked, cbChecked, cbGrayed);

TCheckBox = class(TCustomCheckBox)
    property ALIGNMENT: TAlignment; 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: TCheckBoxState; read write;
    property ONCLICK: TNotifyEvent; read write;
end;

TRadioButton = class(TButtonControl)
    property ALIGNMENT: TALIGNMENT; read write;
    property CAPTION: String; read write;
    property CHECKED: BOOLEAN; read write;
    property COLOR: TColor; read write;
    property FONT: TFont; read write;
    property ONCLICK: TNotifyEvent; read write;
    property ONDBLCLICK: TNotifyEvent; read write;
end;

TCustomListBox = class(TWinControl)
    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(TCustomListBox)
    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: TListBoxStyle; read write;
    property ONCLICK: TNotifyEvent; read write;
    property ONDBLCLICK: TNotifyEvent; read write;
    property ONKEYDOWN: TKeyEvent; read write;
    property ONKEYPRESS: TKeyPressEvent; read write;
    property ONKEYUP: TKeyEvent; read write;
end;

TBevelShape = (bsBox, bsFrame, bsTopLine, bsBottomLine, bsLeftLine,
bsRightLine,bsSpacer);

TBevelStyle = (bsLowered, bsRaised);

```

```

TBevel = class(TGraphicControl)
    property SHAPE: TBEVELSHAPE; read write;
    property STYLE: TBEVELSTYLE; 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: TPanelBevel; read write;
    property BEVELWIDTH: TBevelWidth; read write;
    property BORDERWIDTH: TBorderWidth; read write;
    property BORDERSTYLE: TBorderStyle; 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(TWinControl)
    property AUTOSIZE: BOOLEAN; read write;
    property CAPTION: String; read write;
    property COLOR: TColor; read write;
    property FOCUSCONTROL: TWinControl; read write;
    property FONT: TFont; read write;
    property SHOWACCELCHAR: Boolean; read write;
    property WORDWRAP: Boolean; read write;
    property ONCLICK: TNotifyEvent; read write;
    property ONDBLCLICK: TNotifyEvent; read write;
end;

TNewCheckBox = class(TCustomListBox)
    function AddCheckBox(const ACaption, ASubItem: string; ALevel: Byte;
AChecked, AEnabled, AHasInternalChildren, ACheckWhenParentChecked: Boolean;
AObject: TObject): Integer;
    function
ADDGROUP(ACAPTION, ASUBITEM: STRING; ALEVEL: BYTE; AOBJECT: TObject): INTEGER;
    function AddRadioButton(const ACaption, ASubItem: string; ALevel: Byte;
AChecked, AEnabled: Boolean; AObject: TObject): Integer;
    property CHECKED[Index: INTEGER]: BOOLEAN; read write;
    property STATE[Index: INTEGER]: TCHECKBOXSTATE; 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 ALLOWGRAYED: BOOLEAN; read write;
    property FLAT: BOOLEAN; read write;

```

```

property MINITEMHEIGHT: INTEGER; read write;
property OFFSET: INTEGER; read write;
property MULTISELECT: BOOLEAN; read write;
property ONCLICKCHECK: TNOTIFYEVENT; read write;
property BORDERSTYLE: TBORDERSTYLE; read write;
property COLOR: TColor; read write;
property FONT: TFont; read write;
property SORTED: Boolean; read write;
property STYLE: TListBoxStyle; read write;
property ONCLICK: TNotifyEvent; read write;
property ONDBLCLICK: TNotifyEvent; read write;
property ONKEYDOWN: TKeyEvent; read write;
property ONKEYPRESS: TKeyPressEvent; read write;
property ONKEYUP: TKeyEvent; read write;
property SHOWLINES: BOOLEAN; read write;
property WANTTABS: BOOLEAN; read write;
end;

TNewProgressBar = class(TWinControl)
  property MIN: LONGINT; read write;
  property MAX: LONGINT; read write;
  property POSITION: LONGINT; read write;
end;

TRichEditViewer = class(TMemo)
  property RTFTEXT: STRING; write;
  property USERICHEDIT: BOOLEAN; read write;
end;

TPasswordEdit = class(TCustomEdit)
  property AUTOSELECT: Boolean; read write;
  property AUTOSIZE: Boolean; 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 Password: Boolean; read write;
  property READONLY: Boolean; read write;
  property TEXT: string; read write;
  property ONCHANGE: TNotifyEvent; read write;
  property ONCLICK: TNotifyEvent; read write;
  property ONDBLCLICK: TNotifyEvent; read write;
  property ONKEYDOWN: TKeyEvent; read write;
  property ONKEYPRESS: TKeyPressEvent; read write;
  property ONKEYUP: TKeyEvent; read write;
end;

TCustomFolderTreeView = class(TWinControl)
  procedure ChangeDirectory(const Value: String; const CreateNewItem:
Boolean);
  procedure CreateNewDirectory(const ADefaultName: String);
  property Directory: String; read write;
end;

TFolderRenameEvent = procedure(Sender: TCustomFolderTreeView; var NewName:
String; var Accept: Boolean);

```

```

TFolderTreeView = class(TCustomFolderTreeView)
  property OnChange: TNotifyEvent; read write;
  property OnRename: TFolderRenameEvent; read write;
end;

TStartMenuFolderTreeView = class(TCustomFolderTreeView)
  procedure SetPaths(const AUserPrograms, ACommonPrograms, AUserStartup,
ACommonStartup: String);
  property OnChange: TNotifyEvent; read write;
  property OnRename: TFolderRenameEvent; 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;
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(TCustomControl)
  property Color: TColor; read write;
  property Notebook: TNewNotebook; 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(TComponent)
  property ID: Integer; read;
  property Caption: String; read write;
  property Description: String; read write;
  property Surface: TNewNotebookPage; read write;
  property SurfaceHeight: Integer; read write;
  property SurfaceWidth: Integer; read write;
  property OnActivate: TWizardPageNotifyEvent; read write;
  property OnBackButtonClick: TWizardPageButtonEvent; read write;
  property OnCancelButtonClick: TWizardPageCancelEvent; read write;
  property OnNextButtonClick: TWizardPageButtonEvent; read write;
  property OnShouldSkipPage: TWizardPageShouldSkipEvent; read write;
end;

```

```

TInputDialogWizardPage = class(TWizardPage)
    function Add(const APrompt: String; const APassword: Boolean): Integer;
    property Edits[Index: Integer]: TPasswordEdit; read;
    property PromptLabels[Index: Integer]: TNewStaticText; read;
    property SubCaptionLabel: TNewStaticText; read;
    property Values[Index: Integer]: String; read write;
end;

TInputDialogOptionWizardPage = class(TWizardPage)
    function Add(const ACaption: String): Integer;
    function AddEx(const ACaption: String; const ALevel: Byte; const
AExclusive: Boolean): Integer;
    property CheckListBox: TNewCheckListBox; read;
    property SelectedValueIndex: Integer; read write;
    property SubCaptionLabel: TNewStaticText; read;
    property Values[Index: Integer]: Boolean; read write;
end;

TInputDialogDirWizardPage = class(TWizardPage)
    function Add(const APrompt: String): Integer;
    property Buttons[Index: Integer]: TButton; 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;

TInputDialogFileWizardPage = class(TWizardPage)
    function Add(const APrompt, AFilter, ADefaultExtension: String): Integer;
    property Buttons[Index: Integer]: TButton; 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;

TInputDialogMsgWizardPage = class(TWizardPage)
    property MsgLabel: TNewStaticText; read;
end;

TInputDialogMsgMemoWizardPage = class(TWizardPage)
    property RichEditViewer: TRichEditViewer; read;
    property SubCaptionLabel: TNewStaticText; read;
end;

TInputDialogProgressWizardPage = class(TWizardPage)
    procedure Hide;
    property Msg1Label: TNewStaticText; read;
    property Msg2Label: TNewStaticText; read;
    property ProgressBar: TNewProgressBar; read;
    procedure SetProgress(const Position, Max: Longint);
    procedure SetText(const Msg1, Msg2: String);
    procedure Show;
end;

TInputDialogStateForm = class(TForm)
end;

```

```

TSetupForm = class(TUIStateForm)
  procedure Center;
  procedure CenterInsideControl(const Ctl: TWinControl; const
InsideClientArea: Boolean);
end;

TMainForm = class(TSetupForm)
  procedure ShowAboutBox;
end;

TWizardForm = class(TSetupForm)
  property CANCELBUTTON: TBUTTON; read;
  property NEXTBUTTON: TBUTTON; read;
  property BACKBUTTON: TBUTTON; read;
  property NOTEBOOK1: TNOTEBOOK; read;
  property NOTEBOOK2: TNOTEBOOK; read;
  property WelcomePage: TNewNotebookPage; read;
  property InnerPage: TNewNotebookPage; read;
  property FinishedPage: TNewNotebookPage; read;
  property LicensePage: TNewNotebookPage; read;
  property PasswordPage: TNewNotebookPage; read;
  property InfoBeforePage: TNewNotebookPage; read;
  property UserInfoPage: TNewNotebookPage; read;
  property SelectDirPage: TNewNotebookPage; read;
  property SelectComponentsPage: TNewNotebookPage; read;
  property SelectProgramGroupPage: TNewNotebookPage; read;
  property SelectTasksPage: TNewNotebookPage; read;
  property ReadyPage: TNewNotebookPage; read;
  property PreparingPage: TNewNotebookPage; read;
  property InstallingPage: TNewNotebookPage; read;
  property InfoAfterPage: TNewNotebookPage; read;
  property DISKSPACELABEL: TNewStaticText; read;
  property DIREEDIT: TEDIT; read;
  property GROUPEEDIT: TEDIT; read;
  property NOICONSHECK: TCHECKBOX; read;
  property PASSWORDLABEL: TNewStaticText; read;
  property PASSWORDEDIT: TPasswordEdit; read;
  property PASSWORDEDITLABEL: TNewStaticText; read;
  property READYMEMO: TMEMO; read;
  property TYPESCOMBO: TCOMBOBOX; read;
  property BEVEL: TBEVEL; read;
  property WizardBitmapImage: TBitmapImage; read;
  property WELCOMELABEL1: TNewStaticText; read;
  property INFOBEFOREMEMO: TRICHEDITVIEWER; read;
  property INFOBEFORECLICKLABEL: TNewStaticText; read;
  property MAINPANEL: TPANEL; read;
  property BEVEL1: TBEVEL; read;
  property PAGENAMELABEL: TNewStaticText; read;
  property PAGEDESCRIPTIONLABEL: TNewStaticText; read;
  property WizardSmallBitmapImage: TBitmapImage; read;
  property READYLABEL: TNewStaticText; read;
  property FINISHEDLABEL: TNewStaticText; read;
  property YESRADIO: TRADIOBUTTON; read;
  property NORADIO: TRADIOBUTTON; read;
  property WizardBitmapImage2: TBitmapImage; read;
  property WELCOMELABEL2: TNewStaticText; read;

```

```
property LICENSELABEL1: TNewStaticText; read;
property LICENSEMEMO: TRICHEDITVIEWER; read;
property INFOAFTERMEMO: TRICHEDITVIEWER; read;
property INFOAFTERCLICKLABEL: TNewStaticText; read;
property COMPONENTSLIST: TNEWCHECKLISTBOX; read;
property COMPONENTSDISKSPACE LABEL: TNewStaticText; read;
property BEVELED LABEL: TNewStaticText; read;
property STATUS LABEL: TNewStaticText; read;
property FILENAME LABEL: TNewStaticText; read;
property PROGRESSGAUGE: TNEWPROGRESSBAR; read;
property SELECTDIR LABEL: TNewStaticText; read;
property SELECTSTARTMENUFOLDER LABEL: TNewStaticText; read;
property SELECTCOMPONENTS LABEL: TNewStaticText; read;
property SELECTTASKS LABEL: TNewStaticText; read;
property LICENSEACCEPTEDRADIO: TRADIOBUTTON; read;
property LICENSENOTACCEPTEDRADIO: TRADIOBUTTON; read;
property USERINFONAME LABEL: TNewStaticText; read;
property USERINFONAMEEDIT: TEDIT; read;
property USERINFOORGLABEL: TNewStaticText; read;
property USERINFOORGEDIT: TEDIT; read;
property PreparingErrorBitmapImage: TBitmapImage; read;
property PREPARING LABEL: TNewStaticText; read;
property FINISHEDHEADING LABEL: TNewStaticText; read;
property USERINFOSERIAL LABEL: TNewStaticText; read;
property USERINFOSERIALEDIT: TEDIT; read;
property TASKSLIST: TNEWCHECKLISTBOX; read;
property RUNLIST: TNEWCHECKLISTBOX; read;
property DirBrowseButton: TButton; read;
property GroupBrowseButton: TButton; read;
property SelectDirBitmapImage: TBitmapImage; read;
property SelectGroupBitmapImage: TBitmapImage; read;
property SelectDirBrowseLabel: TNewStaticText; read;
property SelectStartMenuFolderBrowseLabel: TNewStaticText; read;
property CurPageID: Integer; read;
function ADJUST LABEL HEIGHT (ALABEL: TNewStaticText): INTEGER;
procedure INCTOP DEC HEIGHT (ACONTROL: TCONTROL; AMOUNT: INTEGER);
end;
```

