Overview

The Setup API provides a set of functions that your setup application can call to perform installation operations.

These setup functions work closely with Windows® 95- and Windows NT® 4.0- style INF files to provide the following setup functionality.

For information about…	**See…**
Queueing files	File Queues
Installing files	File Queues INF Files Creating Setup Applications
Handling errors and prompting for media	Disk Prompting and Error Handling
Updating registry entries	INF Files
Logging installed files	File Log
Storing the most recently used source paths	MRU Source List

Unicode and ANSI versions are available for most setup functions. You can use the ANSI setup functions to create setup applications for the Windows 95 operating system. For Windows NT setup applications, you can use either ANSI or Unicode versions.

Note Unicode text files should contain the standard 0XFEFF byte-order mark to enable setup functions to identify the file as Unicode text.

Although the Setup API supports prompting for new media and basic error-handling dialog boxes, the setup functions do not provide wizard functionality or a generic user interface.

Setup Applications

Typically, before you create a setup application, you create an *INF file*. An INF file is a text file that contains information used by your setup application during an installation. For more information, see INF Files.

After creating an INF file, you write the source code for your setup application. You can use the setup functions in your setup application to access the information in the INF file and perform installation operations. For more information about using the setup functions in an application, see Creating Setup Applications.

File Version Comparisions

If the SP_COPY_NEWER flag is specified during a file copy operation, the setup functions check for an existing copy of the file in the target directory. If an existing copy is found, the functions compare the versions of the target and source file to determine which is newer.

The file version information used during version checks is that specified in the **dwFileVersionMS** and **dwFileVersionLS** members of a **VS_FIXEDFILEINFO** structure, used by the Win32 version functions. If one of the files does not have version resources specified or if they have the same version information, the source file is treated as the newer file.

INF Files

Information (INF) files store information in a specific format. The setup functions retrieve information from the INF file to use when performing installation operations.

Examples of the type of information stored in an INF file include INI and registry changes, filenames, and locations of the source files on source media.

About INF Files

The following topics describe the new cross-platform INF file format used for Windows 95 and Windows NT 4.0. This new format replaces the script-based INF files used with Windows NT 3.*x*.

Also discussed are the types of sections in the new INF files, and how they fit together to provide the information that the setup functions use during an installation.

Introducing INF Files

Before you develop a Windows NT or Windows 95 setup application, you will probably want to create an information (INF) file.

An INF file is made up of different types of sections. Each type of section has a particular purpose; for example, to copy files or to add entries to the registry. Each line in a section contributes to that section's purpose. The number and type of sections used in an INF file depends on the installation procedure.

Each section type has a particular format. The setup functions use this format to retrieve information from the INF file. This format is described in the following topics.

To be used by the operating system installer, an INF section must contain one or more items.

In addition to the formatted INF sections, an INF file can contain private sections. You can use a private section to store specialized information required by your setup application. The format of a private section will depend on your application.

The Setup API includes low-level functions that you can use to retrieve information from these private sections. Low-level setup functions, such as the **SetupGetLineText** and **SetupGetIntField** functions, retrieve information from the INF file at the line and field level.

Additional sections types are used when installing device drivers. For more information about these other sections, see the *DDK Programmer's Guide*.

INF File Format

The new INF file format contains many types of named sections. Some of these are only used when installing device drivers and are not described in this document. For more information about these sections, see the *DDK Programmer's Guide*.

When you create an INF file, you do not have to include every type of section. The sections you use will depend on your installation procedure.

The setup functions use the following INF sections.

Add Registry	Contains information used to add subkeys or value names to the registry.
Copy Files	Contains source filenames and can specify additional copying behavior.
Delete Registry	Contains information used to delete a subkey or value name from the registry.
Delete Files	Contains the filenames of files to delete, and can specify additional deletion behavior.
DestinationDirs	Maps **Copy Files** sections to a destination directory.
EventLog Install	Contains information used to add an event message to the registry.
INI File to Registry	Contains information used to move lines or sections from an INI file to the registry.
Install	Contains a list of the INF sections that the setup functions process during the installation.
Rename Files	Contains the source and target filenames for renaming operations.
Service Install	Contains information used to install the service listed in the AddService key of the **Services** section.
Services	Lists the services to add or install.
SourceDisksFiles	Maps the source files to their source location.
SourceDisksNames	Assigns an ordinal value to the source disks and can store additional information about the source disks.
Strings	Maps string keys (values in the INF file surrounded by percent signs (%)) to replacement strings.
Update INI Fields	Contains information used to replace, add, or delete fields in an INI entry.
Update INI File	Contains information used to replace, add, or delete an INI entry.
Version	Contains information about the INF file.

The only section that must be included is the **Version** section. The setup functions use this section to recognize an INF file as formatted for Windows 95 and Windows NT.

Each section title in the INF file, must be enclosed by square brackets ([ ]). In general, you can specify the title for an INF section. The exceptions to this are the **DestinationDirs**, **SourceDisksFiles**, **SourceDisksNames**, **Strings**, and **Version** sections. The titles of these sections must be identical to the section type. For example:

```
[DestinationDirs]
```

The **SourceDisksNames** and **SourceDisksFiles** sections can be specified with a platform-specific suffix. The following example shows an Intel-specific **SourceDisksNames** section.

```
[SourceDisksNames.x86]
```

The setup functions programmatically determine the platform of the user's system. If the setup functions find platform-specific **SourceDisksNames** and **SourceDisksFiles** sections that match the user's platform, the setup functions use the platform-specific sections. Otherwise the setup functions use the default (non-suffixed) **SourceDisksNames** and **SourceDisksFiles** sections.

The formatting of lines in a section varies according with its type. For more information about the format of a particular section, see INF File Format Reference.

INF File Sections

The different sections of an INF file fit together to provide the information used by the setup functions during an installation. The central section that defines the steps of the installation is the **Install** section.

The **Install** section provides an overview of the installation process. Each line of an **Install** section has two parts. On the left of the equals sign (=) is the key. On the right hand side, is a list of one or more section titles. The key specifies the type of the sections that are listed on the right.

To better understand this, consider the following example of an **Install** section.

```
[MyInstallSection]
Copyfiles=DataFiles, ProgramFiles
Delfiles=OldFiles
UpdateInis=NewIniInfo
AddReg=NewRegistryInfo, MoreNewRegistryInfo
DelReg=OldRegistryInfo, MoreOldRegistryInfo
```

In the preceding example, the **CopyFiles** key is given the values *DataFiles* and *ProgramFiles*. These specify two **Copy Files** sections in the INF file that contain the source filenames for the copying operations necessary for the installation. You can specify one or more **Copy Files** sections for the **CopyFiles** key of an **Install** section.

Simliarly, the **Delfiles** key specifies **Delete Files** sections that contain information relevant to file deletion operations. The **UpdateInis** key specifies **Update INI File** sections that contain information about updating entries in the INI file, and the **AddReg** and **DelReg** keys specify **Add Registry** and **Delete Registry** sections that contain information about adding or deleteing registry information.

For more information about the types of sections that can be specified in an **Install** section, see the INF File Format Reference.

The **DestinationDirs** section defines the target directory for files listed in **Copy Files**, **Rename Files**, or **Delete Files** sections.

The **SourceDisksNames** section assigns an ordinal value to each source disk. You can store additional information about the source disks, such as a human-readable description in this section.

The **SourceDisksFiles** section maps the source files to the ordinal values assigned in the **SourceDisksNames** section.

You can have multiple platform-specific **SourceDisksNames** and **SourceDisksFiles** sections in an INF file. You add a platform-specific suffix to the title of a **SourceDisksNames** or **SourceDisksFiles** section to indicate that the information listed in that section is platform-specific.

If the setup functions find platform-specific **SourceDisksNames** and **SourceDisksFiles** sections that match the user's platform, the setup functions use the platform-specific sections. Otherwise the setup functions use the default (non-suffixed) **SourceDisksNames** and **SourceDisksFiles** sections.

For example, consider the following sections from an INF file.

```
[SourceDisksNames]
1="NT CD", \default

[SourceDisksNames.mips]
1="NT CD", \mips

[SourceDisksNames.alpha]
```

```
 1="NT CD", \alpha
```

If the user's machine was a MIPS-based system, the setup functions use the information listed in the section titled **SourceDisksNames.mips** and look for source files in the \mips directory of the specified source media.

For an Intel-based system, no platform-specific section exists, and the setup functions use the information listed in **SourceDisksNames** and look for source files in the \default directory.

The **Strings** section maps strings keys, values used as place-holders in an INF file and enclosed by percent signs (%), to the printable strings they represent. You can use strings keys as placeholders in an INF file for information that changes frequently or needs to be localized.

INF File Format Reference

The following sections describe the syntax and meaning of the items used in each type of INF file section. INF files must follow these general rules:

- Sections begin with the section name enclosed in brackets.
- A **Version** section must be included in any INF file formatted for Windows 95 and Windows NT 4.0. The **Version** section contains information about the INF file itself.
- Values may be expressed as replaceable strings using the form %*strkey*%. To use a % character in the string, use %%. The strkey must be defined in a **Strings** section of the INF file.

The following INF sections can be used with the setup functions to create an installation application. For information about INF sections used to install device drivers see the *DDK Programmer's Guide*.

Add Registry

Copy Files

Delete Registry

Delete Files

DestinationDirs

EventLog Install

INI File to Registry

Install

Rename Files

Service Install

Services

SourceDisksFiles

SourceDisksNames

Strings

Update INI Fields

Update INI File

Version

Add Registry Overview

Group

The **Add Registry** section adds subkeys or value names to the registry, optionally setting the value. The *add-registry-section* name must appear in an **AddReg** item in an Install section.

```
[add-registry-section]
reg-root-string, [subkey], [value-name], [flags], [value]
[reg-root-string, [subkey], [value-name], [flags], [value]]
.
.
.
```

reg-root-string

Registry root name. This parameter can be one of the following values.

HKCR	Same as **HKEY_CLASSES_ROOT**.
HKCU	Same as **HKEY_CURRENT_USER**.
HKLM	Same as **HKEY_LOCAL_MACHINE**.
HKU	Same as **HKEY_USERS**.
HKR	Relative to the key passed into SetupInstallFromInfSection.

subkey

Optional. Identifies the subkey to set. Has the form *key1\key2\key3*....

value-name

Optional. Identifies the value name for the *subkey*. For string type, if the *value-name* parameter is left empty, the value of the subkey is set to the default value for that registry entry.

flags

Optional. Establishes the value data type and the **AddReg** item action. The flag value is a bitmap where the low word contains basic flags that define the general data type and **AddReg** item action. The high word contains values that more specifically identify the data type of the registry value. The high word is ignored by the 16-bit Windows 95 setup functions in SETUPX.DLL. The flags are defined as follows:

Value	**Meaning**
FLG_ADDREG_BINVALUETYPE	The value is "raw" data.
FLG_ADDREG_NOCLOBBER	Do not overwrite the registry key if it already exists.
FLG_ADDREG_DELVAL	Delete the value from the registry.
FLG_ADDREG_APPEND	Append a value to an existing value. This flag is currently supported only for REG_MULTI_SZ values.
FLG_ADDREG_TYPE_MASK	Mask.
FLG_ADDREG_TYPE_SZ	Registry data type REG_SZ.
FLG_ADDREG_TYPE_MULTI_SZ	Registry data type REG_MULTI_SZ.
FLG_ADDREG_TYPE_EXPAND_SZ	Registry data type REG_EXPAND_SZ.
FLG_ADDREG_TYPE_BINARY	Registry data type REG_BINARY.

FLG_ADDREG_TYPE_DWORD	Registry data type REG_DWORD.
FLG_ADDREG_TYPE_NONE	Registry data type REG_NONE.

Bit 0 of the flag distinguishes between character and binary data as it does in the Windows 95 setup functions, thus a Windows 95 installation program will see the extended data types as REG_SZ or REG_BINARY. To allow REG_DWORD entries to be compatible with both operating systems, the following formats are supported.

- Non compatible format. If compatibility with Windows 95 setup functions is not required, a REG_DWORD entry can contain a single data value field. This value can be prefixed with a sign and can be either decimal or hexadecimal. For example:

  ```
  HKLM,"Software\Microsoft\Windows
  NT\CurrentVersion\FontDPI","LogPixels",0x10001,120
  ```

- Windows 95-compatible format. If compatibility with Windows 95 setup functions is required, the data of a FLG_ADDREG_TYPE_DWORD entry must be formatted like REG_BINARY. The Windows NT setup functions recognize a REG_DWORD line with exactly four data elements as compatible with Windows 95. The setup functions interpret the four data elements as one **DWORD**. Hexadecimal number fields are only supported by the Windows 95 setup functions as members of a REG_BINARY data list, in which case the data is assumed to be in hexadecimal format (the 0x prefix must not be used). The previously listed example can be written in Windows 95-compatible format as follows:

  ```
  HKLM,"Software\Microsoft\Windows NT\CurrentVersion\FontDPI","LogPixels",
  65537,78,0,0,0
  ```

To represent a number with a data type other than the predefined REG_* types, you can specify the type number in the flag's high word and specify binary type in its low word. You must enter the data in binary format, one byte per field. For example, to store 16 bytes of data with a new data type of 0x38, you would have an **AddReg** item as follows:

```
HKR,,MYValue,0x00380001,0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
```

You can use this technique for any data type except REG_EXPAND_SZ, REG_MULTI_SZ, REG_NONE, and REG_SZ.

value

Optional. Value to set. This parameter can be either a string or a number in hexadecimal notation. At least two fields are required; however, one can be null. Therefore, at least one comma is required when using this form.

The two items in the following example **Add Registry** section, which is named sermouse_EventLog_AddReg, add two value names to the registry, EventMessageFile and TypesSupported, and set the value of these names.

```
[sermouse_EventLog_AddReg]
HKR,,EventMessageFile,0x00020000,"%%SystemRoot%%\System32\IoLogMsg.dll;%
%SystemRoot%%\System32\drivers\sermouse.sys"
HKR,,TypesSupported,0x00010001,7
```

Copy Files Overview

Group

A **Copy Files** section lists the files to copy from a source disk to a destination directory. The source disk and destination directory associated with each file are specified in other sections of the INF file. The *file-list-section* name must appear in the **CopyFiles** item of an **Install** section.

You can copy a single file with the **CopyFiles** item of an **Install** section, without building a **Copy Files** section. For more information, see Install Section.

```
[file-list-section]
destination-file-name[,source-file-name][,temporary-file-name][,flag]
[destination-file-name[,source-file-name][,temporary-file-name]][,flag]
.
.
.
```

destination-file-name
Name of the destination file. If no source filename is given, this is also the name of the source file.

source-file-name
Name of the source file. If the source and destination filenames for the file copy operation are the same, *source-file-name* is not required.

temporary-file-name
This value is ignored. The setup functions automatically generate names for temporary files.

flag
Optional. These flags can be used to control how files are copied. You must specify the actual numerical value in the INF file.

COPYFLG_WARN_IF_SKIP (0x00000001)
Display a warning if the user tries to skip a file after an error has occurred.

COPYFLG_NOSKIP (0x00000002)
Do not allow the user to skip copying the file.

COPYFLG_NOVERSIONCHECK (0x00000004)
Ignore file versions and write over existing files in the destination directory.

COPYFLG_FORCE_FILE_IN_USE (0x00000008)
Force file-in-use behavior. Handle the file as if it was in use during the file copying operation.

COPYFLG_NO_OVERWRITE (0x00000010)
Do not overwrite an existing file in the destination directory.

COPYFLG_NO_VERSION_DIALOG (0x00000020)
Do not overwrite a file in the destination directory if the existing file is newer than the source file.

COPYFLG_REPLACEONLY (0x00000040)
Copy the source file to the destination directory only if the file is already present in the destination directory.

The following example copies two files:

```
[CopyTheseFilesSec]
file11                          ; copies file11
file31, file32                  ; copies file32 to file31
```

All the source filenames used in this example must be defined in a **SourceDisksFiles** section and the Directory identifiers that appear in the **SourceDisksFiles** section must be defined in a **SourceDisksNames** section. As an alternative, you can use a layout INF file specified in the **Version** section to supply this information. A layout INF file is a file that contains a **SourceDisksFiles** section and a **SourceDisksNames** section.

Delete Registry

Overview
Group

A **Delete Registry** section deletes a subkey or value name from the registry. The *del-registry-section* name must appear in a **DelReg** item in an **Install** section.

A **Delete Registry** section can contain any number of items. Each item deletes one subkey or value name from the registry.

```
[del-registry-section]
reg-root-string, subkey, [value-name]
[reg-root-string, subkey, [value-name]]
.
.
.
```

reg-root-string
Registry root name. This parameter can be one of the following values:

HKCR	Same as **HKEY_CLASSES_ROOT**.
HKCU	Same as **HKEY_CURRENT_USER**.
HKLM	Same as **HKEY_LOCAL_MACHINE**.
HKU	Same as **HKEY_USERS**.
HKR	Relative from the key passed into **SetupInstallFromInfSection**.

subkey
Identifies the subkey to delete. Has the form *key1\key2\key3*... This parameter can be expressed as a replaceable string. For example, you could use %Subkey1% where the string to replace %Subkey1% is defined in the **Strings** section of the INF file.

value-name
Optional. Identifies the value name for the *subkey*. The *value-name* parameter can be expressed as a replaceable string. For example, you could use %Valname1% where the string to replace %Valname1% is defined in the **Strings** section of the INF file.

Delete Files Overview

Group

A **Delete Files** section lists the files to be deleted. The *file-list-section* name must appear in the **Delfiles** item of an **Install** section.

```
[file-list-section]
file-name[,,,flag]
.
.
.
```

file-name

Identifies a file to be deleted. *File-name* must be defined in a **SourceDisksFiles** section.

flag

Optional. These flags can be used to control how files are copied. You must specify the actual numerical value in the INF file.

DELFLG_IN_USE (0x00000001)

If the file is in use when SetupCommitFileQueue is called, queue the file to be deleted when the system is rebooted.

Note If you do not use this flag along with a *file-name* parameter and the file is in use when the **Delete Files** section is executed, the file will not be deleted from the system.

DELFLG_IN_USE1 (0x00010000)

This flag is a high-word version of the DELFLG_IN_USE. Setting DELFLG_IN_USE1 causes the same behavior as setting DELFLG_IN_USE with the difference that it enables you to use the same section as both a **Copy Files** and **Delete Files** section without causing conflicts if the COPYFLG_WARN_IF_SKIP flag is used.

The following example deletes three files:

```
[DeleteOldFilesSec]
file1
file2
file3
```

DestinationDirs Overview

Group

The **DestinationDirs** section defines the destination directories for the operations specified in file-list sections (**Copy Files**, **Rename Files**, or **Delete Files**). As an option, you can specify a default destination directory for any **Copy Files**, **Rename Files**, or **Delete Files** sections in the INF file that are not explicitly named in the **DestinationDirs** section.

```
[DestinationDirs]
file-list-section=drid[,subdir]
.
.
.
[DefaultDestDir=drid[,subdir]]
```

file-list-section

Name of a **Copy Files**, **Rename Files, or Delete Files** section. Typically, you will refer to this section in the **CopyFiles**, **RenFiles**, or **DelFiles** line of an **Install** section.

drid

A directory identifier (DRID). The installer replaces a DIRID with an actual name during installation.

A DIRID has the form *%dirid%* where *dirid* is one of the predefined identifiers or an identifier defined in the **DestinationDirs** section. When using a DIRID, you should use the backslash in a path. For example, *%11%\card.ini* can be used to reference *card.ini* in the *system32* directory.

A DIRID can be one of the following values:

-01, 0xffff	The directory from which the INF was installed.
01	SourceDrive:\path.
10	Windows directory.
11	System directory. (%windir%\system on Windows 95, %windir%\system32 on Windows NT)
12	Drivers directory.(%windir%\system32\drivers on Windows NT)
17	INF file directory.
18	Help directory.
20	Fonts directory.
21	Viewers directory.
24	Applications directory.
25	Shared directory.
30	Root directory of the boot drive.
50	%windir%\system
51	Spool directory.
52	Spool drivers directory.
53	User Profile directory.
54	Path to ntldr or OSLOADER.EXE

subdir

Name of the directory, within the directory named by *drid*, to be the destination directory.

The optional **DefaultDestDir** item provides a default destination for any **CopyFiles** items that use the direct copy (@filename) notation or any **Copy Files**, **Rename Files**, or **Delete Files** sections not specified in the **DestinationDirs** section. If a **DefaultDestDir** is not used in a **DestinationDirs** section, the default directory is set to *drid_system*.

This example sets the default directory for all the sections of the INF file to the drivers directory:

```
[DestinationDirs]
DefaultDestDir = 12 ; DIRID_DRIVERS
```

EventLog Install

Overview

Group

The **EventLog Install** section adds an event message file to the registry. You can also use an **EventLog Install** section to remove an event message file.

```
[install-section-name_EventLogInstallSection]
AddReg=add-registry-section
DelReg=del-registry-section
```

add-registry-section
: The section of the INF file that specifies subkeys to add to the registry to install the event message file.

del-registry-section
: The section of the INF file that specifies subkeys to delete from the registry to remove the event message file.

The following is an example of a typical **EventLog Install** section:

```
[sermouse_EventLog_Inst]
AddReg=sermouse_EventLog_AddReg

[sermouse_EventLog_AddReg]
HKR,EventMessageFile,0x00020000,"%%SystemRoot%%\System32\IoLogMsg.dll\;%
%SystemRoot%%\System32\drivers\sermouse.sys"
HKR,,TypesSupported,0x00010001,7
```

INI File to Registry

Overview

Group

The **INI File to Registry** section moves lines or sections from an INI file to the registry, creating or replacing a registry entry under the specified key in the registry. The section name *ini-to-registry-section* must appear in an **Ini2Reg** item in an **Install** section of the INF file.

```
[ini-to-registry-section]
ini-file, ini-section, [ini-key],reg-root-string,subkey[,flags]
.
.
.
```

ini-file
: Name of the INI file containing the key to copy. For more information about specifying the INI filename, see the **Update Ini File** section.

ini-section
: Name of the section in the INI file containing the key to copy.

ini-key
: Name of the key in the INI file to copy to the registry. If *ini-key* is empty, the whole section is transferred to the specified registry key.

reg-root-string
: Registry root name. This parameter can be one of the following values:

HKCR	Same as **HKEY_CLASSES_ROOT**.
HKCU	Same as **HKEY_CURRENT_USER**.
HKLM	Same as **HKEY_LOCAL_MACHINE**.
HKU	Same as **HKEY_USERS**.
HKR	Relative from the key passed into **SetupInstallFromInfSection**.

subkey
: Identifies the subkey to receive the value. Has the form *key1\key2\key3*...

flags
: Indicates whether to delete the INI key after transfer to the registry and whether to overwrite the value in the registry if the registry key already exists. This parameter can be one of the following values.

Bit	**Value**	**Meaning**
0	0	(Default) Do not delete the INI entry from the INI file after moving the information in the entry to the registry.
0	1	Delete the INI entry from the INI file after moving the information in the entry to the registry.
1	0	(Default) If the registry subkey exists, do not replace its current value.
1	1	If the registry subkey exists, replace its current value with the value from the INI file entry.

Install

Overview

Group

The **Install** section identifies the sections in the INF file that contain instructions for installing files.

```
[install-section-name]
LogConfig=log-config-section-name[,log-config-section-name]...
Copyfiles=file-list-section[,file-list-section]...
Renfiles=file-list-section[,file-list-section]...
Delfiles=file-list-section[,file-list-section]...
UpdateInis=update-ini-section[,update-ini-section]...
UpdateIniFields=update-inifields-section[,update-inifields-section]...
AddReg=add-registry-section[,add-registry-section]...
DelReg=del-registry-section[,del-registry-section]...
Ini2Reg=ini-to-registry-section[,ini-to-registry-section]...
```

Not all the types of items shown in the syntax are needed or required in an **Install** section. Section names must consist of printable characters. An item can contain more than one section ; each name after the first must be preceded with a comma. Items must specify the name of the corresponding section in the INF file. The only exception to this is the **CopyFiles** item, which need not specify a section if only one file is to be copied.

> **Note** You only use **Log Config** sections when installing a device driver. For more information, see the *DDK Programmer's Guide*.

You can use a special notation in the **CopyFiles** item to copy a single file directly from the CopyFiles line. You can copy individual by prefixing the filename with an @ symbol. The destination for any file copied using this notation is the **DefaultDestDir** item, as defined in a **DestinationDirs** section. This is demonstrated in the following example.

```
[MyInstall]
CopyFiles=@MyFile.exe
```

By appending an extension to the name of the **Install** section, you can have different **Install** sections for different operating systems or platforms, if necessary. The Setup routines recognize the following extensions:

.Win	Windows 95
.NT	Windows NT (all platforms)
.NTx86	Windows NT (x86 only)
.NTMIPS	Windows NT (MIPS only)
.NTAlpha	Windows NT (Alpha only)
.NTPPC	Windows NT (PPC only)

The following example **Install** section consists of a single line that lists two **Copy Files** sections:

```
[Ser_Inst]
CopyFiles=Ser_CopyFiles, mouclass_CopyFiles

[Ser_CopyFiles]
sermouse.sys

[mouclass_CopyFiles]
mouclass.sys
```

Rename Files

Overview

Group

A **Rename Files** section lists the names of files to be renamed. The name of the section must appear in a **Renfiles** item in an **Install** section of the INF file.

```
[rename-files-section-name]
new-file-name,old-file-name
.
.
.
```

new-file-name
New name of the file.

old-file-name
Old name of the file. The *old-file-name* parameter must be defined in a **SourceDisksFiles** section.

The following example renames file42 to file41, file52 to file51, and file62 to file61:

```
[RenameOldFilesSec]
file41, file42
file51, file52
file61, file62
```

Service Install

Overview

Group

The **Service Install** section installs the service listed in the **AddService** entry of the **Services** section:

```
[install-section-name_ServiceInstallSection]
DisplayName=[name]
ServiceType=type-code
StartType=start-code
ErrorControl=error-control-level
ServiceBinary=path-to-service
LoadOrderGroup=[load-order-group-name]
Dependencies=+depend-on-group-name[[,depend-on-service-name]...]
StartName=[driver-object-name]
```

name

Optional. A friendly name for the service.

type-code

Specifies the type of driver. This can be any type allowed by the **CreateService** function.

start-code

Specifies when to start the driver. This parameter can be one of the following values:

SERVICE_BOOT_START (0x0)

Indicates a driver started by the operating system loader. Use this code only for drivers essential to loading the operating system.

SERVICE_SYSTEM_START (0x1)

Indicates a driver started during system initialization.

SERVICE_AUTO_START (0x2)

Indicates a driver started by the Service Control Manager during system startup.

SERVICE_DEMAND_START (0x3)

Indicates a driver that the Service Control Manager starts on demand.

SERVICE_DISABLED (0x4)

Indicates a driver that cannot be started.

error-control-level

Specifies the level of error control. This parameter can be one of the following values:

CRITICAL (0x3)

If the driver fails to load, fail the attempted startup. If the startup is not using the LastKnownGood control set, switch to LastKnownGood. If the startup attempt is using LastKnownGood, run a bug-check routine.

SEVERE (0x2)

If the startup is not using the LastKnownGood control set, switch to LastKnownGood. If the startup attempt is using LastKnownGood, continue on in case of error.

NORMAL (0x1)

If the driver fails to load or initialize, startup should proceed, but display a warning.

IGNORE (0x0)

If the driver fails to load or intialize, startup proceeds. No warning is displayed.

path

The path to the binary for the service.

load-order-group-name

Optional. Identifies the load order group of which this driver is a member. Examples of groups are: pointer, port, primary disk, and so forth.

+depend-on-group-name

Optional. Specifies a load order group on which the driver depends. The driver is started only if at least one member of the specified group has been started.

depend-on-service-name

Optional. Specifies a service that must be running before this driver is started.

driver-object-name

Optional. If *type-code* specifies a Kernel driver or a file system driver, this name is the Windows NT driver object name that the I/O Manager uses to load the device driver.

The following example shows a typical **Service Install** section:

```
[mouclass_Service_Inst]
DisplayName    = %mouclass.SvcDesc%
ServiceType    = 1                  ; SERVICE_KERNEL_DRIVER
StartType      = 1                  ; SERVICE_SYSTEM_START
ErrorControl   = 1                  ; SERVICE_ERROR_NORMAL
ServiceBinary  = %12%\mouclass.sys
LoadOrderGroup = Pointer Class
```

Services

Overview

Group

The **Services** section lists services to add to a computer. You can remove services by including a **DelService** entry in the **Services** section.

```
[install-section-name.Services]
AddService=ServiceName,flag,service-install-section[,event-log-install-
section]
DelService=ServiceName
```

ServiceName
The service to install or delete.

flag
Specifies how to add the service. This parameter is only used with the AddService key and can be one of the following flags.

SPSVCINST_TAGTOFRONT (0x1)
Move the service's tag to the front of its group order list.

SPSVCINST_ASSOCSERVICE (0x2)
Associate the service in this **AddService** item with the device to be installed by this INF file.

service-install-section
The name of the **Service Install** section.

event-log-install-section
The name of the **EventLog Install** section.

The following example shows a typical **Services** section:

```
[Ser_Inst.Services]
AddService = sermouse, 0x00000002, sermouse_Service_Inst,
sermouse_EventLog_Inst ; Port Driver
AddService = mouclass,, mouclass_Service_Inst, mouclass_EventLog_Inst
                       ; Class Driver
```

SourceDisksFiles

Overview

Group

The **SourceDisksFiles** section names the source files used during installation and identifies the source disks that contain the files. The source disks are listed in **SourceDisksNames**.

In order to allow multi-platform distribution of source files, you can construct a platform-specific **SourceDisksFiles** section. For example, on a MIPS platform, all setup functions that use a **SourceDisksFiles** section will first look for a **SourceDisksFiles.Mips** section. If the setup functions do not find a **SourceDisksFiles.Mips** section, or the specified line is not found in that section, the setup functions will use the **SourceDisksFiles** section. This behavior affects any setup function that directly or indirectly references a **SourceDisksFiles** section as part of its processing.

The suffixes are not case sensitive and can be one of the following.

- alpha
- mips
- ppc
- x86

```
[SourceDisksFiles]
filename=disk-number[,subdir][,size]
.
.
.
```

filename
Name of the file on the source disk.

disk-number
Ordinal of the source disk that contains the file. This ordinal must be defined in a **SourceDisksNames** section, and must have a value greater than or equal to 1.

subdir
Optional parameter that specifies the subdirectory on the source disk where the file resides. If this parameter is not used, the source disk root directory is the default.

size
Optional parameter that specifies the uncompressed size of the file.

The following example **SourceDisksFiles** section identifies a single source file, SRS01.x86, on the disk with the ordinal of 1.

```
[SourceDisksFiles]
SRS01.x86 = 1
```

SourceDisksNames Overview

Group

The **SourceDisksNames** section identifies and names the disk that contains the source files for file copying and renaming operations. The source files are listed in **SourceDisksFiles**.

In order to allow multi-platform distribution of source files, you can construct a platform-specific **SourceDisksNames** section. For example, on a MIPS platform, all setup functions that use a **SourceDisksNames** section will first look for a **SourceDisksNames.Mips** section. If the setup functions do not find a **SourceDisksNames.Mips** section, or the specified line is not found in that section, the setup functions will use the **SourceDisksFiles** section. This behavior affects any setup function that directly or indirectly references a **SourceDisksFiles** section as part of its processing.

The suffixes are not case sensitive and can be one of the following.

- alpha
- mips
- ppc
- x86

```
[SourceDisksNames]
disk-ordinal="disk-description",disk-label,unused[,path]
.
.
.
```

disk-ordinal
A unique number that identifies a source disk. If there is more than one source disk, each must have a unique ordinal.

disk-description
A string or strings key describing the content or purpose of the disk. The installer displays this string to the user to identify the disk. The description must be enclosed in double quotation marks.

disk-label
The volume label of the source disk that is set when the source disk is formatted.

unused
This parameter is not used.

path
Optional parameter that specifies the absolute path to the source files. If this parameter is not used, the root directory is the default.

In the following example, the *write.exe* file is the same for all platforms and is located in the *\common* directory on the compact disc. The *disk-number* of 1 directs the setup functions to the *\common* directory, regardless of which platform the files are being installed on. The CMD.EXE file is a platform-specific file that is distributed for all platforms. The *disk-number* of 2 directs the setup functions to the correct directory for each platform. The HALNECMP.DLL file is specific to the MIPS platform.

```
[SourceDisksNames]
1 = "Windows NT CD-ROM", Instd1,, \common

[SourceDisksNames.Alpha]
2 = "Windows NT CD-ROM", Instd1,, \alpha

[SourceDisksNames.Mips]
2 = "Windows NT CD-ROM", Instd1,, \mips
```

```
[SourceDisksNames.x86]
2 = "Windows NT CD-ROM", Instd1,, \x86

[SourceDisksNames.ppc]
2 = "Windows NT CD-ROM", Instd1,, \ppc

[SourceDisksFiles]
write.exe = 1
cmd.exe = 2

[SourceDisksFiles.Mips]
halnecmp.dll = 2
```

Strings Overview

Overview

The **Strings** section defines one or more strings keys. A strings key is a name that represents a string of printable characters. Although the **Strings** section is usually the last section in an INF file, a strings key defined in a **Strings** section can be used anywhere in the INF file that uses the string. The installer expands the string key to the specified string and uses it for further processing. When a strings key is used, it must be enclosed by percent signs (%). To use a % character in the string, use %%.

```
[Strings]
strings-key=value
.
.
.
```

strings-key
: A unique name consisting of letters and digits.

value
: A string consisting of letters, digits, or other printable characters. It should be enclosed in double quotation marks if the corresponding strings key is used in a type of item that requires double quotation marks.

The **Strings** section simplifies translation of strings for international markets by placing all strings that can be displayed at the user interface in a single section of the INF file. Strings keys should be used whenever possible. The following example shows the **Strings** section of a typical INF file.

```
[Strings]
String0="Corporation X"
String1="Corporation X"
String2="CS2590 SCSI Adapter"
```

Update INI Fields Overview

Overview

The **Update INI Fields** section replaces, adds, and deletes fields in the value of an INI entry. Unlike an **Update INI File** section, an **Update INI Fields** section replaces, adds, or deletes portions of a value in an INI file entry rather than the whole value. The section name, *update-inifields-section-name*, must appear in the **UpdateIniFields** item in an **Install** section of the INF file.

```
[update-inifields-section-name]
ini-file,ini-section,profile-name,[old-field],[new-field],[flags]
.
.
.
```

ini-file
Name of the INI file containing the entry to change. For more information about specifying the INI filename, see Update INI File Section.

ini-section
Name of the INI file section containing the entry to change.

profile-name
Name of the entry to change.

old-field
Field value to delete.

new-field
Field value to add, if it does not exist.

flags
Specifies whether to treat the *old-field* and *new-field* arguments as though they have a wild card character. The parameter specifies which separator character to use when appending a new field to an INI file entry. The *flags* argument can be any of the following values.

Bit	Value	Meaning
0	0	(Default) Treat the asterisk (*) character literally and not as a wild card character when matching fields .
0	1	Treat the asterisk (*) character as a wild card character when matching fields.
1	0	(Default) Use a blank ( ) as a separator when adding a new field to an entry.
1	1	Use a comma (,) as a separator when adding a new field to an entry.

The setup functions remove any comments in the INI file line because they might not be applicable after changes. When looking for fields in the line in the INI file, the setup functions search for spaces, tabs, and commas as field delimiters. However, a space is used as the separator when the new field is appended to the line.

Update INI File Overview

Overview

The **Update INI File** section replaces, deletes, or adds complete entries in the INI file. The section name, *update-ini-section-name*, must appear in the **UpdateInis** item in the **Install** section of the INF file.

```
[update-ini-section-name]
ini-file,ini-section,[old-ini-entry],[new-ini-entry],[flags]
.
.
.
```

ini-file
: The INI file containing the entry to change.

ini-section
: Name of the section containing the entry to change.

old-ini-entry
: Optional. Usually in the form *key*=*value*.

new-ini-entry
: Optional. Usually in the form *key*=*value*. Either the key or value may specify replaceable strings. For example, either the key or value specified in the *new-ini-entry* parameter may be %String1%, where the string that replaces %String1% is defined in the **Strings** section of the INF file.

flags
: Optional action flags. The *flags* parameter can be one of the following values:

Value	Meaning
0	Default. If *old-ini-entry* key is present in an INI file entry, that entry is replaced with *new-ini-entry*. Only the keys of the *old-ini-entry* parameter and the INF file entry must match; the value of each entry is ignored. To add *new-ini-entry* to the INI file unconditionally, set *old-ini-entry* to NULL. To delete *old-ini-entry* from the INI file unconditionally, set *new-ini-entry* to NULL.
1	If both key and value of *old-ini-entry* exist in an INI file entry, that entry is replaced with *new-ini-entry*. The *old-ini-entry* parameter and the INF file entry must match on both key and value for the replacement to be made (this is in contrast to using an action flag value of 0, where only the keys must match for the replacement to be made).
2	If the key in the *old-ini-entry* parameter does not exist in the INI file, no operation is performed on the INI file. If the key in the *old-ini-entry* parameter exists in an INI file entry and the key in the *new-ini-entry* parameter exists in an INI file entry, the INI file entry that matches the key in the *new-ini-entry* parameter is deleted, and the key of the INI file entry that matches the *old-ini-entry* parameter is replaced with the key in the *new-ini-entry* parameter. If the key in the *old-ini-entry* parameter exists in an INI file entry and the key in the *new-ini-entry* parameter does not exist in an INI file entry, an entry made up of the key in the *new-ini-entry* parameter and the old value is added to the INI file.

The match of the *old-ini-entry* parameter and an INI file entry is based on key alone, not key and value.

3 Same as the *flags* parameter value of 2 described preceding, except the match of the *old-ini-entry* parameter and an entry in the INF file is based on matching both key and value, not just the key.

You can use the wild card character (*) in specifying the key and value and they will be interpreted correctly.

The *ini-filename* can be a string or a strings key. A strings key has the form *%strkey%* where *strkey* is defined in the **Strings** section in the INF file. The name must be a valid filename.

The *ini-filename* should include the name of the directory containing the file, but the directory name should be given as a directory identifier rather than an actual name. The installer replaces a directory identifier with an actual name during installation.

A directory identifier has the form *%dirid%* where *dirid* is one of the predefined identifiers or an identifier set by **SetupSetDirectoryId**. When using a directory identifier, you should use the backslash in a path. For example, you can use *%11%\card.ini* to reference CARD.INI in the System32 directory.

The following examples illustrate individual items in an **Update INI File** section:

```
%11%\sample.ini, Section1,, Value1=2              ; adds new entry
%11%\sample.ini, Section2, Value3=*,              ; deletes old entry
%11%\sample.ini, Section4, Value5=1, Value5=4     ; replaces old entry
```

Version Overview

Overview

The **Version** section must be included in all INF files formatted for use with Windows 95 and Windows NT 4.0.

```
[Version]
Signature="signature-name"
Class=class-name
ClassGUID=GUID
Provider=INF-creator
LayoutFile=filename.inf[,filename.inf]...
```

signature-name

This parameter can be either $Windows NT$, $Chicago$, or $Windows 95$. If *signature-name* is not one of these strings, the file is not accepted as an INF file for the classes of devices recognized by Windows NT. Signature string recognition is case-insensitive.

class-name

This parameter is only used when installing device drivers. For more information see the *DDK Programmer's Guide*.

GUID

This parameter is only used when installing device drivers. For more information see the *DDK Programmer's Guide*.

INF-creator

Identifies the creator of the INF file. Typically, this is the name of the organization that creates the INF file.

filename.inf

The INF file that contains the SourceDisksFiles and SourceDisksNames sections required for installing this the application. This file is usually named LAYOUT.INF. If *filename.inf* is not specified, the **SourceDisksNames** and **SourceDisksFiles** sections must be included in the current INF file.

The following example shows a typical **Version** section:

```
[Version]
Signature="$Windows NT$"
Class=Mouse
ClassGUID={4D36E96F-E325-11CE-BFC1-08002BE10318}
Provider=%Provider%
LayoutFile=layout.inf

[Strings]
Provider="Corporation X"
```

Using INF Files

After you create an INF file for the installation process, the setup functions can access that information to use when copying, deleting, or renaming files and updating INIs and registry entries.

The following topics describe how to use the setup functions to retrieve information from INF files and how to perform installation operations by referencing an INF file.

Opening and Closing an INF file

Before the setup functions can access information in the INF, you must open it with a call to the **SetupOpenInfFile** function. This function returns a handle to the INF file.

If you do not know the name of the INF file that you need to open, you can use the **SetupGetInfFileList** function to obtain a list of all the INF files of a particular type (either old Windows NT 3.*x* script-based files, or new Windows 95- and Windows NT 4.0-style INF files, or both) in a directory.

After you open an INF file, you can append additional INF files to the opened INF file by using the **SetupOpenAppendInfFile** function. This is functionally similar to an include statement. When subsequent setup functions reference an open INF file, they will also be able to access any information stored in the appended files.

If you do not specify an INF file during the call to the **SetupOpenAppendInfFile** function, **SetupOpenAppendInfFile** appends the file(s) specified by the **LayoutFile** key in the **Version** section of the open INF file.

When you no longer need the information in the INF file, call the **SetupCloseInfFile** functionto release resources allocated during the call to **SetupOpenInfFile**.

Retrieving Information From an INF File

After you have a handle to an INF file, you can retrieve information from it in a variety of ways. Functions such as **SetupGetInfInformation**, **SetupQueryInfFileInformation**, and **SetupQueryInfVersionInformation** retrieve information about the INF file itself.

Other functions such as **SetupGetSourceInfo** and **SetupGetTargetPath** acquire information about the source files and target directories.

Low-level functions like **SetupGetLineText** and **SetupGetStringField** enable you to directly access information stored in a line or field of an INF file. These functions are used internally by the higher-level functions, but are also available should you need to access INF information at the line or field level.

Installing From an INF File

After you retrieve installation information from an INF file, there are several file-handling functions that you can use to install the files listed in an INF section. Low-level functions such as **SetupInstallFile** and **SetupInstallFileEx** install a single file.

There are also functions to handle compressed files. The **SetupGetFileCompressionInfo** function returns information about compressed files. This information can then be used by **SetupDecompressOrCopyFile** to copy and, if necessary, expand the file.

High-level functions such as **SetupInstallFromInfSection**, **SetupInstallFilesFromInfSection**, and **SetupInstallServicesFromInfSection** process the installation operations in an INF section. Of these, **SetupInstallFromInfSection** is the most versatile because it can perform any type of installation operation listed in the **Install** section of an INF file. This includes the registry and INI operations listed in the **AddReg**, **DelReg**, **UpdateInis**, or **UpdateIniField** lines of an **Install** section.

The **SetupInstallFilesFromInfSection** and **SetupInstallServicesFromInfSection** functions queue operations from an INF section to an existing file queue. For more information, see File Queues.

In contrast, the **SetupInstallFromInfSection** function creates and destroys its own internal queue. A common use for **SetupInstallFromInfSection** is to call it after all files have been successfully copied to perform the registry and INI transactions.

In addition to the functions previously listed, the Setup API includes functions that queue file installation operations, either by file, or by INF section. For more information, see File Queues.

INF File Reference

The following topics list the data types, structures, functions and notifications used with INF files.

INF File Data Types

The following data type is used with INF files.

HINF	Handle to a loaded INF file.

INF File Structures

The following structures are used with INF files.

INFCONTEXT	Positional context in an INF file.
SP_INF_INFORMATION	Information about the INF file

Note The structure INFCONTEXT is used internally by the setup functions and must not be altered or referenced by setup applications. It is included here for informational purposes only.

INF File Functions

The following functions are used with INF files.

SetupCloseInfFile	Frees resources and closes the INF handle.
SetupDecompressOrCopyFile	Copies a file and, if necessary, decompresses it.
SetupFindFirstLine	Finds a the first line in a section of an INF file or, if a key is specified, the first line that matches that key. It updates the **Line** member of an INFCONTEXT structure.
SetupFindNextLine	Returns the next line in a section relative to the **Line** member of the specified INFCONTEXT structure.
SetupFindNextMatchLine	Returns the next line in a section relative to the **Line** member of the specified INFCONTEXT that matches a specified key.
SetupGetBinaryField	Retrieves data from a line whose fields are in binary format.
SetupGetFieldCount	Returns the number of fields in a line.
SetupGetFileCompressionInfo	Retrieves file compression information from an INF file.
SetupGetInfFileList	Gets a list of the types of INF files in a specified directory.
SetupGetInfInformation	Returns information about an INF file (by **Line** member of an INFCONTEXT or filename).
SetupGetIntField	Returns the specified integer field of a line in an INF file.
SetupGetLineByIndex	Updates the **Line** member of an INFCONTEXT for the line at a specified index in the specified section.
SetupGetLineCount	Returns the number of lines in the specified section.
SetupGetLineText	Retrieves the content of a specified line from an INF file.
SetupGetMultiSzField	Returns a list of strings, starting at the specified field of a line in an INF file.
SetupGetSourceFileLocation	Gets the source disk ordinal and path (relative to source root) where the source file is located

SetupGetSourceFileSize	Gets the file size for an individual source file or a **Copy Files** section of an INF file.
SetupGetSourceInfo	Retrieves the path, tag file, or description for a source.
SetupGetStringField	Returns the specified string field of a line in an INF file.
SetupGetTargetPath	Gets the target path for a **Copy Files** section in an INF file.
SetupInstallFile	Installs a file.
SetupInstallFileEx	Installs a file and returns a variable indicating whether or not the file was in use.
SetupInstallFilesFromInfSection	Queues all the files in an INF file **Copy Files** section.
SetupInstallFromInfSection	Performs the directives specified in an INF file **Install** section.
SetupInstallServicesFromInfSection	Performs service installation and deletion operations as specified in a **Service Install** section of an INF file.
SetupOpenAppendInfFile	Opens an INF file and append it to an existing INF handle.
SetupOpenInfFile	Opens an INF file and returns a handle to it.
SetupOpenMasterInf	**Windows NT only:** Opens the INF file that contains file and layout information for files shipped with Windows NT.
SetupQueryInfFileInformation	Queries an INF information structure about its associated INF filename(s).
SetupQueryInfVersionInformation	Queries an INF information structure for version information on one of its constituent INF files.
SetupSetDirectoryId	Associates a new directory identifier with a particular directory.

INF File Notifications

The following notifications are used with the **SetupInstallFile**, **SetupInstallFileEx**, and **SetupInstallFromInfSection** functions. For more information about the format and use of notifications, see Notifications.

SPFILENOTIFY_FILEOPDELAYED	The file is in use, and the current operation is delayed until the system is rebooted.
SPFILENOTIFY_LANGMISMATCH	The language of the current file does not match the system language.
SPFILENOTIFY_TARGETEXISTS	A copy of the specified file already exists on the target.
SPFILENOTIFY_TARGETNEWER	A newer version of the specified file exists on the target.

Note Because **SetupInstallFromInfSection** creates and commits an internal file queue, it also uses the File Queue Notifications.

Disk Prompting and Error Handling

There are four setup functions that provide basic functionality to prompt the user for a new source disk, or to create error dialog boxes. The following sections describe these dialog boxes and how to use them in setup applications and in callback routines.

About Disk Prompting and Error Handling

Though the setup functions do not provide a user interface, there are four setup functions that generate dialog boxes to handle common installation situations and gather information from the user. These are: **SetupPromptForDisk**, **SetupCopyError**, **SetupRenameError**, and **SetupDeleteError**.

Callback routines can call these functions to create dialog boxes to aid in processing notifications sent by other setup functions such as **SetupCommitFileQueue** and **SetupInstallFile**.

The **SetupPromptForDisk** function prompts the user to insert removable media, specify a new source path, or cancel the installation. The application can offer additional options to the user, depending on the flags specified when the function is called. These include skipping the current file, or browsing for a new source path.

The three functions, **SetupCopyError**, **SetupRenameError**, and **SetupDeleteError**, create dialog boxes that interact with the user to gather information from the user on how to proceed when an error has occurred.

The **SetupCopyError** function generates a dialog box that asks the user how to recover from a copy error. The user can specify a new source path for the copy operation or cancel the installation. Depending on the flags specified during the call to **SetupCopyError**, the user may also be able to browse for a new source path, view error details, or skip the current file.

A dialog box that asks the user how to process errors that occur during a file renaming operation can be generated by calling **SetupRenameError**. With this dialog box, the user has the opportunity to retry the operation, skip the current rename operation, or abort.

The **SetupDeleteError** function generates a dialog box that can gather input on how the user wishes to handle an error that occurred during a file deletion operation. The user is given the options to retry the operation, skip the current delete operation, or abort.

The default queue callback routine, **SetupDefaultQueueCallback**, uses the previously mentioned four functions to generate parts of its user interface and to handle errors and prompt for new media.

Using Disk Prompting and Error Handling

The dialog boxes created by the setup functions, **SetupPromptForDisk**, **SetupCopyError**, **SetupDeleteError**, and **SetupRenameError** can be used by a setup application, or by callback routines. The following sections describe using the functions in an application, and their corresponding notifications.

Prompting For a Disk

To generate a dialog box that prompts the user to insert the next disk or specify a new source path, call **SetupPromptForDisk**. This function is used by callback routines to generate a user interface when the notification, SPFILENOTIFY_NEEDMEDIA, is sent to the callback routine.

The dialog box generated by **SetupPromptForDisk** gives the user the option to insert a disk, specify a new source path, or cancel the installation.

You can use flags specified during the call to **SetupPromptForDisk** to alter the layout and behavior of the dialog box. Using these flags, you can control whether the dialog box includes buttons that allow the user to browse for a new source path, or skip the current file. The flags also enable you to control dialog box behavior such as beeping when first displayed, and the ability be displayed as a foreground window.

Error Handling

There are three functions that generate dialog boxes to handle errors: **SetupCopyError**, **SetupDeleteError**, and **SetupRenameError**.

Callback routines can use these functions to generate a user interface to handle SPFILENOTIFY_COPYERROR, SPFILENOTIFY_DELETEERROR, and SPFILENOTIFY_RENAMEERROR notifications.

Each of these error-handling functions generates a dialog box that asks the user for information on how to proceed. As with **SetupPromptForDisk**, you can modify the dialog box's layout and behavior by specifying flags during the function call.

Disk Prompting and Error Handling Reference

The following sections list the functions that create dialog boxes for basic disk prompting and error handling.

Disk Prompting and Error Handling Functions

The following functions provide disk prompting and an error handling user interface.

SetupCopyError	Generates a dialog box that informs the user of a copy error.
SetupDeleteError	Generates a dialog box that informs the user of a deletion error.
SetupPromptForDisk	Generates a dialog box that prompts the user for a media disk or source file location.
SetupRenameError	Generates a dialog box that informs the user of a rename error.

File Queues

The setup functions include file queue functionality. A file queue is a list of file copying, renaming, and deletion operations. These operations can be sent to the queue in any order. When the queue is committed, these operations are processed as a batch, in order of the operation type.

The following sections explain what a queue is and how to use it when creating a setup application. Also discussed is the order in which the enqueued file operations are processed as the queue commits and what notifications the queue sends to a callback routine at each stage.

About File Queues

A file queue is a list of file operations that are processed at one time. The file operations in the queue may be copy, rename, or delete operations. The file queue organizes file operations by type, creating copy, rename, and delete subqueues.

These operations may be sent to the queue in any order, and the enqueueing process need not be contiguous.   When the queue is committed, the **SetupCommitFileQueue** function performs file operations in order of the operation type.

Typically, all of the file operations necessary for an entire installation are queued to the file queue, and then processed in a single batch when the queue is committed.

One advantage of queueing file operations over installing files section-by-section from an INF file is that you can streamline the installation process. Instead of having to obtain information from the user for each section to be installed, you can obtain installation information from the user for all the files to be installed while building the queue. This frees the user to pursue other activities while the time-intensive copy operations are processed by the **SetupCommitFileQueue** function.

Another advantage of file queues is that you can track the progress of the installation as a whole. When installing section-by-section from an INF file, progress indicators such as progress bars can track only the current INF section. When the next section is installed, the progress bar starts over. With a queue, the total number of files to be processed during the entire installation is known before the queue is committed, and thus, a progress bar can be generated to track the entire installation.

Order of Queue Commitment

When the **SetupCommitFileQueue** function commits the file queue, it processes the file operations in the following order: file deletion operations, then file renaming operations, and finally, file copying operations. The following figure illustrates the life cycle of a queue's commitment process.

```
start the queue
    start the delete subqueue
        start a file delete operation        < - - repeat for each
        finish a file delete operation       < - - queued file delete
    finish the delete subqueue

    start the rename subqueue
        start a file rename operation        < - - repeat for each
        finish a file rename operation       < - - queued file rename
    finish the rename subqueue

    start the copy subqueue
        start a file copy operation          < - - repeat for each
        finish a file copy operation         < - - queued file copy
    finish the copy subqueue
finish the queue
```

At each step, or if an error occurs, the **SetupCommitFileQueue** function sends a notification to the callback routine. The callback routine can use the information sent by the queue to track the installation progress and, if necessary, interact with the user.

For example, if a file copy operation needed a source file that was not available at the current path, **SetupCommitFileQueue** would send a SPFILENOTIFY_NEEDMEDIA notification to the callback routine, along with information about the file and media required. The callback routine could use this information to generate a dialog box that prompts the user to insert the next disk by calling **SetupPromptForDisk**

A default queue callback routine, **SetupDefaultQueueCallback**, is included with the Setup API. This routine handles queue notifications and generates error dialog boxes and progress bars for the installation. You can use the default queue callback routine as it is, or write a filter callback routine to handle a subset of the notifications and pass the others on to the default queue callback routine.

If none of the functionality of the callback routine suits your needs, you can write a self-contained custom callback routine that does not call the default queue callback routine.

For more information about queue callback routines, see Default Queue Callback Routine, and Creating a Custom Queue Callback Routine.

Queue Notifications

After a queue is committed by calling **SetupCommitFileQueue**, it will begin to process the queued operations. At each step, the queue sends a notification to the callback routine specified in the call to **SetupCommitFileQueue**.

Following is the syntax that **SetupCommitFileQueue** uses to send a notification to the callback routine.

```
MsgHandler(          //the specified callback routine
    Context,         //context used by the callback routine
    Notification,    //queue notification code
    Param1,          //additional notification information
    Param2            //additional notification information
);
```

The values of *Param1* and *Param2* contain additional information relevant to the notification being sent to the callback routine. Each notification, its *Param1* and *Param2* values, and how the default queue callback routine handles that notification, is described in detail in File Queue Notifications.

Using File Queues

The following sections describe how to use the setup functions to create a queue, add file operations to a queue, and commit the queue.

These sections also describe how the setup functions process the platform-specific sections of an INF file and how to override this functionality.

Opening and Closing a Queue

Before you can queue file operations, you must open a file queue. Calling the **SetupOpenFileQueue** function returns a handle to a queue file. This handle is used by subsequent queue functions to reference the open queue and add operations to it or scan it.

After all the specified file operations have been queued and committed, you must call the **SetupCloseFileQueue** function to release resources allocated during the call to **SetupOpenFileQueue**.

> **Note** The **SetupCloseFileQueue** function does not commit the file queue. Any operations that are uncommitted when **SetupCloseFileQueue** is called will not be performed.

Queueing Files

After you have obtained a handle to a file queue by calling the **SetupOpenFileQueue** function, you can add file operations to the queue, either file-by-file, or by queueing all the files in an INF section.

To add a copy operation for an individual file to the file queue, call the **SetupQueueCopy** function. If you want to queue a file copy operation using the default source media and target destinations specified in an INF file, you can call the **SetupQueueDefaultCopy** function.

You can add an individual file delete or rename operation to the open file queue, by calling the **SetupQueueDelete** or **SetupQueueRename** function.

Queueing an INF Section

In addition to queueing file operations individually, you can also queue all the files listed in an INF section.

You can queue all the files listed in a **Copy Files** section of an INF file by calling **SetupQueueCopySection**. For this function to be successful, the **Copy Files** section must be in the proper format and the INF file, or one of its appended files, must contain the **SourceDisksFiles** and **SourceDisksNames** sections.

Similarly, if you want to delete all the files specified in a **Delete Files** section of an INF file, you can call **SetupQueueDeleteSection**. To rename all the files in a **Rename Files** section of an INF file use **SetupQueueRenameSection**.

Committing a Queue

After all the desired file operations have been queued, the queue must be committed. This causes the enqueued file operations to be processed.

To commit the file queue, call the **SetupCommitFileQueue** function, specifying a callback routine. The callback routine will receive notifications from **SetupCommitFileQueue** as the file operations are processed. If you want to use the default queue callback routine, you must first initialize the necessary context by calling either **SetupInitDefaultQueueCallback** or **SetupInitDefaultQueueCallbackEx**. For more information about the default queue callback routine, see Default Queue Callback Routine.

> **Note** **SetupCommitFileQueue** should be called before the queue is closed. Any operations that are uncommitted when **SetupCloseFileQueue** is called will not be performed.

Platform Path Override

After a platform path override is set by a call to **SetupSetPlatformPathOverride**, any setup function that queues file copy operations will examine the final component of the source path. If the final component matches the name of the user's platform, the setup function will replace it with the override string set by **SetPlatformPathOverride**.

For example, when installing printer drivers onto a MIPS server, you might want to install drivers for all supported platforms. Queueing the files normally would install the files specified in the MIPS-dependent sections of the INF file, with source paths such as \\root\source\mips.

To install the files for a second platform, you must call **SetupSetPlatformPathOverride** to set a platform override. The following example shows the specific function call to set the platform to Alpha.

```
 test = SetupSetPlatformPathOverride("alpha");
```

After you set the platform to Alpha, file copy operations sent to the queue with a source path of \\root\source\mips would have their source path changed to \\root\source\alpha. You would repeat this process for each platform of interest.

File Queue Reference

The following sections list the data types, functions, and notifications used with file queues.

File Queue Data Types

The following data type is used with file queues.

HSPFILEQ	Handle to a file queue.

File Queue Functions

The following functions are used with file queues

SetupCloseFileQueue	Terminates the queue. Any remaining transactions are not committed.
SetupCommitFileQueue	Commits all queued transactions.
SetupOpenFileQueue	Initializes and returns a handle to the file queue.
SetupPromptReboot	Prompts the user to reboot his or her computer, if necessary.
SetupQueueCopy	Queues a file copy.
SetupQueueCopySection	Queues the files in an INF **Copy Files** section.
SetupQueueDefaultCopy	Queues the files in an INF **Copy Files** section using the default information specified in an INF file.
SetupQueueDelete	Queues a file deletion.
SetupQueueDeleteSection	Queues the files in an INF **Delete Files** section.
SetupQueueRename	Queues a file rename.
SetupQueueRenameSection	Queues the files in an INF **Rename Files** section.
SetupScanFileQueue	Scans the file queue.
SetupSetPlatformPathOverride	Sets the platform path override.

File Queue Notifications

The following notifications are used with file queues.   For more information about the format and use of notifications, see Notifications.

SPFILENOTIFY_COPYERROR	An error occurred during a file copying operation.
SPFILENOTIFY_DELETEERROR	An error occurred during a file deletion operation.
SPFILENOTIFY_ENDCOPY	A file copying operation has ended.
SPFILENOTIFY_ENDDELETE	A file deletion operation has ended.
SPFILENOTIFY_ENDQUEUE	The queue has finished committing.
SPFILENOTIFY_ENDRENAME	A file rename operation has ended.
SPFILENOTIFY_ENDSUBQUEUE	A subqueue (either copy, rename or delete) has ended.
SPFILENOTIFY_FILEOPDELAYED	The file was in use, and the current operation has been delayed until the system is rebooted.
SPFILENOTIFY_LANGMISMATCH	The language of the current operation does not match the system language.
SPFILENOTIFY_NEEDMEDIA	New source media is needed.
SPFILENOTIFY_QUEUESCAN	A node in the file queue has been scanned. (**SetupScanFileQueue** only)
SPFILENOTIFY_RENAMEERROR	An error occurred during a file renaming operation.
SPFILENOTIFY_STARTCOPY	A file copy operation has started.
SPFILENOTIFY_STARTDELETE	A file delete operation has started.
SPFILENOTIFY_STARTQUEUE	The queue has started to commit.
SPFILENOTIFY_STARTRENAME	A file rename operation has started.
SPFILENOTIFY_STARTSUBQUEUE	A subqueue (either copy, rename or delete) has started.
SPFILENOTIFY_TARGETEXISTS	A copy of the specified file already exists on the target.
SPFILENOTIFY_TARGETNEWER	A newer version of the specified file exists on the target.

Default Queue Callback Routine

Included with the setup functions is a default callback routine, **SetupDefaultQueueCallback**, that you can use to process notifications returned by **SetupCommitFileQueue**.

The following sections discuss the format of the default queue callback routine, using the default queue callback routine with **SetupCommitFileQueue**, and how to create a filter callback routine that builds on the functionality provided by the default queue callback routine.

About the Default Queue Callback Routine

The default queue callback routine handles notifications sent by **SetupCommitFileQueue** in a generic manner. By using the default routine, you get a ready-made user interface to create common setup dialog boxes. It is recommended that you use the default queue callback routine, both for ease of use, and to ensure a consistent appearance and behavior of dialog boxes generated during the installation.

The default callback routine requires a context structure for internal record keeping. In addition, the queue passes additional information relevant to the current notification in a set of parameters, *Param1* and *Param2*.

For example, if the queue sends an SPFILENOTIFY_NEEDMEDIA notification to the default callback routine, *Param1* points to a **SOURCE_MEDIA** structure that contains information about the media needed, and *Param2* points to a character array that can receive new path information from the user.

The default callback routine uses this information to prompt the user to either insert the needed source media, specify a new path, skip copying the current file, or cancel the current operation. The default queue callback routine returns FILEOP_NEWPATH, FILEOP_DOIT , FILEOP_SKIP, or FILEOP_ABORT to the queue, depending on which action the user took.

For information on how the default queue callback routine handles each queue notification, see File Queue Notifications.

Creating a Custom Queue Callback Routine

In addition to using the default queue callback, you can write a custom callback routine. This function must have the same form as those pointed to by **SP_FILE_CALLBACK**. This is useful if you need a callback routine to handle a notification in a manner other than that provided by the default queue callback routine.

If only a small portion of the default queue callback routine's behavior needs to be changed, you can create a custom callback routine to filter the notifications, handling only those that require special behavior and calling **SetupDefaultQueueCallback** for the others.

For example, if you wanted to custom-handle file delete errors, you could create a custom callback function, *MyCallback*. This function would intercept and process SPFILENOTIFY_DELETEERROR notifications, and call the default queue callback function for all other notifications. *MyCallback* returns a value for the delete error notifications. For all other notifications, *MyCallback* passes whatever value the default queue callback routine returned to the queue.

This flow of control is illustrated in the following diagram.

{ewc msdncd, EWGraphic, bsd23560 0 /a "SDK.WMF"}

For an example of a setup application that uses a custom callback function, see the Win32 Code Samples.

Important If the custom callback function calls the default queue callback routine, it must pass the void pointer returned by **SetupInitDefaultQueueCallback** or **SetupInitDefaultQueueCallbackEx** to the default callback routine.

Using the Default Queue Callback Routine

The default queue callback routine can be specified to handle notifications sent by **SetupCommitFileQueue**, or it can be called by a custom callback routine that builds on the functionality of the default queue callback routine.

The following sections explain how to initialize and terminate the context used by a default queue callback routine. The sections also describe how to use the default queue callback routine with **SetupCommitFileQueue** or a custom callback routine.

Initializing and Terminating the Callback Context

Before the default queue callback routine can be used, either by specifying it as the callback routine when committing a file queue, or by calling it from a custom callback routine, it must be initialized.

The **SetupInitDefaultQueueCallback** function builds the context structure that is used by the default queue callback routine. It returns a void pointer to that structure. This structure is essential for the default callback routine's operation and must be passed to the callback routine. You do can this either by specifying the void pointer as the context in a call to **SetupCommitFileQueue**, or by specifying the void pointer as the context parameter when calling **SetupDefaultQueueCallback** from a custom callback routine. This context structure must not be altered or referenced by the setup application.

The **SetupInitDefaultQueueCallbackEx** function also initializes a context for the default queue callback routine, but it specifies a second window to receive a caller-specified progress message each time the queue sends a notification. This enables you to use the default disk prompting and error dialog boxes, and to also embed a progress bar in a second window, for example, in a page of an installation wizard.

Regardless of whether you initialized the context used by the default queue callback routine with **SetupInitDefaultQueueCallback** or **SetupInitDefaultQueueCallbackEx**, after the queued operations have finished processing, call **SetupTermDefaultQueueCallback** to release the resources allocated in initializing the context structure.

Calling the Default Queue Callback Routine

If the default queue callback routine is initialized and specified when **SetupCommitFileQueue** is called, the queue calls the the default queue callback routine internally and you need do nothing more.

If you create a filter callback routine that relies on the default queue callback routine to handle a subset of the queue notifications, your filter callback routine must call **SetupDefaultQueueCallback** explicitly.

Important When you call **SetupDefaultQueueCallback** explicitly, you must pass in the void pointer returned by either **SetupInitDefaultQueueCallback** or **SetupInitDefaultQueueCallbackEx**.

One way to do this is to create a context structure for your custom callback routine that includes as one of its members the void pointer returned by **SetupInitDefaultQueueCallback** or **SetupInitDefaultQueueCallbackEx**.

Default Queue Callback Routine Reference

The following sections list the functions used with the default queue callback routine.

Default Queue Callback Routine Functions

The following functions are used with the default queue callback routine.

SetupDefaultQueueCallback	The default queue callback routine.
SetupInitDefaultQueueCallback	Initializes context information needed by the default queue callback routine.
SetupInitDefaultQueueCallbackEx	Initializes context information needed by the default queue callback routine and specifies an alternate window to display progress messages.
SetupTermDefaultQueueCallback	Releases resources allocated by **SetupInitDefaultQueueCallback** and **SetupInitDefaultQueueCallbackEx**.

Cabinet Files

Cabinet files provide a compact and organized way to store compressed source files. The following topics describe what cabinet files are and how they are processed by the setup functions.

About Cabinet Files

A cabinet is a single file, usually suffixed with .CAB, that stores compressed files in a file library. A compressed file can be spread over several cabinet files. During installation, the setup application decompresses the files stored in a cabinet and copies them to the user's system.

If cabinet files are used, the setup functions use the name of the cabinet as the tag file for the source disk it resides on.

Using Cabinet Files

The setup functions handle cabinets internally. To explicitly enumerate and extract files from a cabinet, you can call the **SetupIterateCabinet** function.

The following topic describes the cabinet processing internal to the setup functions and how to use **SetupIterateCabinet**.

Also discussed are the notifications sent by **SetupIterateCabinet** and the required format of a cabinet callback routine to process those notifications.

Extracting Files from Cabinets

You can extract files from a cabinet in two ways. The first and simplest way is to take advantage of the automatic cabinet processing built into the setup functions.

The installation functions, such as **SetupCommitFileQueue**, **SetupInstallFile**, and **SetupInstallFromInfSection**, check the compression on each file. If the file is in a cabinet, the functions first search for a file of that name outside the cabinet. If found, the functions install the external file, ignoring the file inside the cabinet. This enables you to update a single file inside the cabinet without rebuilding the cabinet.

The setup functions also track which files in a cabinet have been retrieved, so that a file is extracted only once, even if it is installed several times.

The second way to extract files from a cabinet is by using **SetupIterateCabinet**. This function iterates through each file in a cabinet, sending a notification to a callback routine for each file found. The callback routine then returns a value that indicates whether the file should be extracted or skipped.

> **Note** The Setup API does not supply a default callback routine to handle cabinet notifications. If you call **SetupIterateCabinet** explicitly, you must supply a callback routine to process the cabinet notifications that the function returns.

Creating a Cabinet Callback Routine

Because the Setup API does not supply a default cabinet callback routine, you need to supply a routine. The callback routine that the **SetupIterateCabinet** function requires must have the same form as those pointed to by **SP_FILE_CALLBACK**.

Following is the syntax that **SetupIterateCabinet** uses to send a notification to the callback routine.

```
MsgHandler(          //the specified callback routine
    Context,         //context used by the callback routine
    Notification,    //cabinet notification
    Param1,          //additional notification information
    Param2            //additional notification information
);
```

The *Context* parameter is a void pointer to a context variable or structure that can be used by the callback routine to store information that needs to persist between subsequent calls to the callback routine.

This context's implementation is specified by the callback routine, and it is never referenced or altered by **SetupIterateCabinet**.

The *Notification* parameter is an unsigned integer and will be one of the following values.

SPFILENOTIFY_FILEEXTRACTED	The file has been extracted from the cabinet.
SPFILENOTIFY_FILEINCABINET	A file is encountered in the cabinet.
SPFILENOTIFY_NEEDNEWCABINET	The current file is continued in the next cabinet.

The final two parameters, *Param1* and *Param2*, are also unsigned integers and contain additional information relevant to the notification. For more information about the notifications sent by **SetupIterateCabinet**, see Cabinet File Notifications.

A SP_FILE_NOTIFY_CALLBACK routine returns an unsigned integer. The cabinet callback routine should return one of the following values depending on the notification.

For the SPFILENOTIFY_FILEINCABINET notification, **SetupIterateCabinet** expects one of the following values to be returned by the callback routine.

FILEOP_ABORT	Abort cabinet processing.
FILEOP_DOIT	Extract the current file.
FILEOP_SKIP	Skip the current file.

For SPFILENOTIFY_NEEDNEWCABINET and SPFILENOTIFY_FILEEXTRACTED notifications, **SetupIterateCabient** expects one of the following values to be returned by the callback routine.

NO_ERROR	No error was encountered, continue processing the cabinet.
ERROR_XXX	An error of the specified type occurred. The **SetupIterateCabinet** function will return FALSE, and the specified error code will be returned by a call to **GetLastError**.

If the callback routine returns FILEOP_DOIT, the routine must also provide a full target path. For more information see SPFILENOTIFY_FILEINCABINET.

Cabinet File Reference

The following sections list the structures, functions, and notifications used with cabinet files.

Cabinet File Structures

The following structures are used with cabinets.

CABINET_INFO	Cabinet file information.
FILE_IN_CABINET_INFO	Information about a file in a cabinet.

Cabinet File Functions

The following function is used with cabinets.

SetupIterateCabinet	Returns a notification for each file stored in a cabinet file.

Cabinet File Notifications

The following notifications are sent by **SetupIterateCabinet**. For more information about the format and use of notifications, see Notifications.

SPFILENOTIFY_FILEEXTRACTED	The file has been extracted from the cabinet.
SPFILENOTIFY_FILEINCABINET	A file is encountered in the cabinet, does the application want to extract it?
SPFILENOTIFY_NEEDNEWCABINET	The current file is continued in the next cabinet.

MRU Source List

The Setup API provides functions that store the most recently used (MRU) source directories. This information is stored on the user's system and can be accessed by subsequent installations.

The following topics describe the MRU source list and how to use the setup functions to open, modify, scan, and close it.

About the MRU Source List

The most recently used (MRU) source list remains resident on the user's system. The setup functions internally handle the creation of new source lists and their location on the user's machine. The setup functions use this list to store information about source paths used in previous installations.

You can use this information when prompting the user for a source path. For example, you could create a drop-down list of the network connections used as source paths in previous installations.

Depending on your permissions, you can create a user list, one that is specific to a particular user, or a system list, one that is the same for all users. In addition to system and user source lists you can create a temporary source list that is discarded when the setup application exits.

Using the MRU Source List

The **SetupSetSourceList** function will open or create a source list on the user's system. You can specify to set the user list, the system list, a combination of the user and system lists, or a temporary list as the MRU source list. If a temporary list is used, it will be the only list available to the setup application until **SetupCancelTemporarySourceList** is called, or **SetupSetSourceList** is called a second time.

After a list is set, you can query the source list by using **SetupQuerySourceList** to obtain an array of the source paths. When the source list array is no longer needed, you must call the **SetupFreeSourceList** function to free the resources allocated by **SetupQuerySourceList**.

To add a path to a source list, either one that is resident on the user's system, or a temporary list, call **SetupAddToSourceList**. If the source list specified is not temporary, that source will remain on the user's system and is accessible to subsequent installations.

To remove a path from the source path, call the **SetupRemoveFromSourceList** function.

MRU Source List Reference

The following topics list the setup functions that provide MRU functionality.

MRU Source Functions

The following functions are used with MRU source file lists.

SetupAddToSourceList	Adds an entry to a source list.
SetupCancelTemporarySourceList	Cancels any temporary list or no-browse behavior and reestablishes standard list behavior.
SetupFreeSourceList	Frees resources allocated to a source list.
SetupQuerySourceList	Queries the current list of installation sources.
SetupRemoveFromSourceList	Removes an entry from an installation source list.
SetupSetSourceList	Sets the installation source list to the system MRU list, the user MRU list, or a temporary list.

File Log

The file log is resident on the user's system and stores information about the files copied during an installation. The following topics describe the types of file logs and how to use the setup functions to open, modify, and close them.

About the File Log

A file log records information about the files copied to a system during an installation. It can be used for emergency repair or as a diagnostic tool.

The file used to log file installations can either be the system log, the file used by the system to store a record of the files installed with the Windows NT operating system, or any other file log that you specify.

Using the File Log

Before you can use a file log, you must call **SetupInitializeFileLog** to open or create it. When you call this function, you can specify flags to create or overwrite a file log, open the system log, or open a file log as read-only.

After **SetupInitializeFileLog** returns a handle to a file log, you can add an entry by calling **SetupLogFile**, delete an entry by calling **SetupRemoveFileLogEntry**, or retrieve information about a particular file in a file log by calling **SetupQueryFileLog**.

When the file log is no longer needed, **SetupTerminateFileLog** should be called to release the resources allocated during the call to **SetupInitializeFileLog**.

File Log Reference

The following sections lists the setup functions that provide file log functionality.

File Log Functions

The following functions are used with file logs.

SetupInitializeFileLog	Initializes a log file for use.
SetupLogFile	Adds an entry to the log file.
SetupQueryFileLog	Retrieves information from a log file.
SetupRemoveFileLogEntry	Removes an entry from a log file.
SetupTerminateFileLog	Releases resources allocated to a log file.

Creating Setup Applications

After you create an INF file, you will typically write the source code for your setup application. You call the setup functions from your setup application to perform many installation operations.

The following sections discuss the steps of a typical installation, and provide specific examples of how you can use the setup functions to perform stage of the installation.

For an example of a complete setup application that implements the following setup procedure, see DOINST.C, included with the Win32 Code Samples.

Steps of an Installation Program

The following list outlines one way to use setup functions to create a setup application. You can use this example as a starting point, or develop your own installation algorithm.

Each step is described in detail in following topics.

Initialization

- Open the INF and, if appropriate, append other INF files.
- Extract file information from the INF file.
- Gather setup information from the user.
- Create a queue.

Install files

- Commit the queue, specifying a callback routine.
- Update registry information.

Clean up

- Close the file queue and INF.
- Release any other system resources and end the program.

Opening the INF File

You must use the **SetupOpenInfFile** function to open the INF file before you can retrieve information from it, or append other INF files to it.

The following example opens an INF file.

```
HINF MyInf;                 //variable to hold the INF handle
PUINT ErrorLine;            //variable to point to errors returned
BOOL test;                  //variable to receive function success

MyInf = SetupOpenInfFile (
      szInfFileName,        //the filename of the inf file to open
      NULL,                 //optional class information
      INF_STYLE_WIN4,       //the inf style
      ErrorLine             //line number of the syntax error
);
```

In the preceding example, *MyInf* is the handle returned by **SetupOpenInfFile** to the opened INF file. The parameter *szInfFileName* specifies the name of the INF file to open. The INF class is specified as NULL. This indicates that the **Class** key should be ignored. The INF_STYLE_WIN4 value specifies that the INF file is formatted in the INF format used with Windows 95 and Windows NT 4.0. The *ErrorLine* parameter is a pointer to a variable that receives the line number of an error that the **SetupOpenInfFile** function generates.

After an INF file is opened, you can call the **SetupOpenAppendInfFile** function to append a file to the open INF file. To append several files, repeat this process.

If you call the **SetupOpenAppendInfFile** function and the filename passed to it is NULL, then the function will search the **Version** section of the open INF file (and any appended INF files) for a **LayoutFile** key. If the function finds a key, it will append the file specified by that key (usually LAYOUT.INF). When multiple INF files have been combined, **SetupOpenAppendInfFile** starts with the last appended INF file when it searches for a **Version** section.

The following example appends the *szSecondInfFileName* file to the open file, *szInfFileName*.

```
test = SetupOpenAppendInfFile (
      szSecondInfFileName,     //name of the inf file to append
                               //to the open inf file, if NULL,
                               //the fn searches for the LayoutInf
                               //key in the version section, and
                               //appends the file specified there.
      MyInf,                   //handle of the open inf file
      ErrorLine                //pointer to an unsigned integer that
                               //receives error information
);
```

In the example, *szSecondInfFileName* is the name of the file to append to the open INF file. *MyInf* is the handle to the open INF file returned by the previous call to the **SetupOpenInfFile** function. The parameter *ErrorLine* points to a variable that will receive any error information generated by the **SetupOpenAppendInfFile** function.

Extracting File Information from the INF file

After the INF file is opened, you can gather information from it to build the user interface, or to direct the installation process. The setup functions provide several levels of functionality for gathering information from an INF file.

To gather information…	Use these functions…
About the INF file	SetupGetInfInformation SetupQueryInfFileInformation SetupQueryInfVersionInformation.
About source and target files	SetupGetSourceFileLocation SetupGetSourceFileSize SetupGetTargetPath SetupGetSourceInfo
From a line of an INF file	SetupGetLineText SetupFindNextLine SetupFindNextMatchLine SetupGetLineByIndex SetupFindFirstLine
From a field of a line in an INF file	SetupGetStringField SetupGetIntField, SetupGetBinaryField SetupGetMultiSzField

The following example uses the **SetupGetSourceInfo** function to retrieve the human-readable description of a source media from an INF file.

```
test = SetupGetSourceInfo (
     MyInf,                    \\Handle to the INF file to access
     SourceId,                 \\Id of the source media
     SRCINFO_DESCRIPTION,      \\which information to retrieve
     Buffer,                   \\a pointer to the buffer to
                               \\  receive the information
     MaxBufSize,               \\the size allocated for the buffer
     &BufSize                  \\buffer size actually needed
);
```

In the example, *MyInf* is the handle to the open INF file. *SourceId* is the identifier for a specific source media. The value SRCINFO_DESCRIPTION specifies that the **SetupGetSourceInfo** function should retrieve the source media description. *Buffer* points to a string that will receive the description, *MaxBufSize* indicates the resources allocated to the buffer, and *BufSize* indicates the resources necessary to store the buffer.

If *BufSize* is greater than MaxBufSize, the function will return FALSE, and a subsequent call to GetLastError will return ERROR_INSUFFICIENT_BUFFER.

Gathering Setup Information from the User

It is highly recommended that your setup application include a user interface that acquires any necessary information from the user. This functionality is not supplied by the setup functions and must be implemented by your setup application, usually in the form of an installation wizard.

Creating a Queue and Queueing File Operations

Queuing the file operations is useful because it enables you to process the installation as a whole, instead of by INF section.

To create a file queue, declare a variable to store the queue handle, then call the **SetupOpenFileQueue** function.

```
HSPFILEQ MyQueue;                    \\variable to hold the queue
                                     \\handle
MyQueue = SetupOpenFileQueue();      \\create the queue
```

In the example, *MyQueue* is the handle to the queue created by **SetupOpenFileQueue**.

After the queue is created, you can queue copy, rename, and delete operations, as well as scan the file queue to verify enqueued operations.

To add single file operations to the queue, use the **SetupQueueCopy**, **SetupQueueRename**, and **SetupQueueDelete** functions.

All the file operations listed in a **Copy Files**, **Delete Files**, or **Rename Files** section can be added to the queue by using **SetupQueueCopySection**, **SetupQueueDeleteSection**, or **SetupQueueRenameSection**, respectively.

Another way to queue all the files in a **Copy Files** section of an INF is to use the function, **SetupInstallFilesFromInfSection**.

The following example uses the **SetupQueueCopySection** function to enqueue copy operations for all the files listed in a **Copy Files** section of an INF file.

```
test = SetupQueueCopySection(
     MyQueue,                  \\Handle of the open queue
     "A:\",                    \\Source root path
     MyInf,                    \\Inf containing the source info
     NULL,                     \\specifies that MyInf contains
                               \\  the section to copy as well
     MySection,                \\the name of the section to queue

                               \\flags specifying the copy style
     SP_COPY_NO_SKIP | SP_COPY_NO_BROWSE,
);
```

In the example, *MyQueue* is the queue to add copy operations to, *"A:\"* specifies the path to the source, and *MyInf* is the handle to the open INF file. The parameter *ListInfHandle* is set to NULL, indicating that the section for copying is in *MyInf*. *MySection* is the section in *MyInf* containing the files to queue for copying.

The flags SP_COPY_NO_SKIP and SP_COPY_NO_BROWSE have been combined using an OR operator to indicate that the user should not be offered options to skip or browse for files if errors occur.

Committing the Queue

If the default callback function is going to be called during the queue commitment, the context for it must be initialized using the **SetupInitDefaultQueueCallback** or **SetupInitDefaultQueueCallbackEx** functions. If you are using a custom callback function that never calls the default callback function, this step is not necessary.

```
PVOID Context;          \\variable to store the context returned

                        \\initialize the context
Context = SetupInitDefaultQueueCallback( OwnerWindow );
```

In the example, *OwnerWindow* is the handle to the window that is to become the parent window of any dialog boxes that the default callback function generates.

After the queue is built and the callback function that will process queue notifications has been initialized, you can call **SetupCommitFileQueue** to commit the operations that have been enqueued.

The following example uses **SetupCommitFileQueue** to commit the queue using the default callback routine.

```
test = SetupCommitFileQueue (
     OwnerWindow,           //window that will own dialog boxes
                            //created by the callback routine
     MyQueue,               //the queue to commit

                            //use the default callback routine
     SetupDefaultQueueCallback

     Context                //context information that will be
                            //  used by the callback routine
);
```

In the preceding example, *MyQueue* is the queue to commit, *OwnerWindow* is the window that will own any dialog boxes created by the default callback routine, *SetupDefaultQueueCallback* specifies that the default callback function will be used, and *Context* is a pointer to the structure returned by the previous call to **SetupInitDefaultQueueCallback**.

Updating Registry Information

After the queue has successfully committed, you will need to update registry information for the product you are installing. It is reccommended that you wait until all necessary file copy operations have been successfully completed before altering registry information.

One way to update the registry is to call **SetupInstallFromInfSection** with the SPINST_INIFILES, SPINST_REGISTRY, or SPINST_INI2REG flags specified. These flags can be combined in one call to **SetupInstallFromInfSection**.

The following example uses SPINST_ALL^SPINST_FILES to indicate that the function should process all of the listed operations except file operations. Since only INI, registry, and file operations are listed in the **Install** section, this is a shorthand method of specifying the function should process all INI and registry operations.

The following example shows how to install registry information using the **SetupInstallFromINFSection** function.

```
Test = SetupInstallFromINFSection (
     NULL,                      \\Window to own any dialog boxes
                                \\  created
     MyInf,                     \\INF file containing the section
     MySection,                 \\the section to install
     SPINST_ALL ^ SPINST_FILES,\\which installation operations
                                \\  to process
     NULL,                      \\the relative root key
     "A:\",                     \\the source root path
     0,                         \\copy style
     NULL,                      \\Message handler routine
     NULL,                      \\Context
     NULL,                      \\Device info set
     NULL                       \\device info data
);
```

In the example, *OwnerWindow* is NULL because only file operations generate dialog boxes, and thus a parent window is not needed. *MyInf* is the INF file containing the section to process. The parameter, *MySection*, specifies the section to be installed. The combined flags, SPINST_ALL ^ SPINST_FILES, specify which installation operations to process, in this case, all operations except file operations. The source root path is specified as *"A:\"*.

Since no copy operations are being processed, the *CopyFlags*, *MsgHandler*, *Context*, *DeviceInfoSet*, and *DeviceInfoData* parameters are not specified.

Closing the File Queue and INF File

After the queue has finished committing its operations, it should be closed so that resources allocated to the queue can be released.

```
SetupCloseFileQueue(MyQueue);             //close file queue
```

Where *MyQueue* is the handle to the queue created by **SetupOpenFileQueue**.

If a default context was initiated for use by the default callback routine, it should also be terminated by calling **SetupTermDefaultQueueCallback**.

```
SetupTermDefaultQueueCallback(Context);   //release default context
                                          //  resources
```

In the example *Context* is a pointer to the structure returned by the **SetupInitDefaultQueueCallback** function.

When access to the INF information is no longer needed, call the **SetupCloseInfFile** function to free system resources.

```
SetupCloseInfFile(MyInf);                 //close inf file
```

MyInf is the handle to the open INF file returned by the **SetupOpenInfFile** function.

Releasing Other System Resources

If a source list or log file was used, resources should be released by calling **SetupFreeSourceList** or **SetupTerminateFileLog**.

Reference

The following sections describe in detail the data types, structures, functions, and notifications of the Setup API.

Data Types

The following data types are used by the setup functions.

HINF	Handle to a loaded INF file.
HSPFILELOG	Handle to a log file.
HSPFILEQ	Handle to a setup file queue.
SP_FILE_CALLBACK	Pointer to a callback routine.

SP_FILE_CALLBACK Quick Info

Overview

Overview

[New - Windows NT]

The **SP_FILE_CALLBACK** data type is a pointer to a callback routine that has the format expected by the setup functions. For information on how to build a callback routine, see Notifications, Creating a Custom Queue Callback Routine, and Creating a Cabinet Callback Routine.

```
typedef UINT (*PSP_FILE_CALLBACK)(
    PVOID Context,          //context used by the callback routine
    UINT Notification,      //notification sent to callback routine
    UINT Param1,            //additional notification information
    UINT Param2             //additional notification information
);
```

Members

Context
Supplies context information about the queue notification being returned to the callback function.

Notification
Specifies the occurrence that triggered the call to the callback function.

Param1
Specifies additional notification information; the value is dependent on which notification is being returned.

Param2
Specifies additional notification information; the value is dependent on which notification is being returned.

See Also

SetupCommitFileQueue, **SetupInstallFile**, **SetupInstallFileEx**, **SetupInstallFromInfSection**, **SetupIterateCabinet**

Structures

The following structures are used by the setup functions.

CABINET_INFO
FILE_IN_CABINET_INFO
FILEPATHS
INFCONTEXT
SOURCE_MEDIA
SP_INF_INFORMATION

CABINET_INFO Quick Info

Overview

Overview

[New - Windows NT]

The **CABINET_INFO** structure stores information about a cabinet file. The **SetupIterateCabinet** function specifies this structure as a parameter when it sends a SPFILENOTIFY_NEEDNEWCABINET notification to the cabinet callback routine.

```
typedef struct _CABINET_INFO {
    PCTSTR CabinetPath;
    PCTSTR CabinetFile;
    PCTSTR DiskName;
    USHORT SetId;
    USHORT CabinetNumber;
} CABINET_INFO, *PCABINET_INFO;
```

Members

CabinetPath
The path to the cabinet file.

CabinetFile
The name of the cabinet file.

DiskName
The name of the source media that contains the cabinet file.

SetId
The identifier of the current set. This number is generated by the software that builds the cabinet.

CabinetNumber
The number of the cabinet. This number is generated by the software that builds the cabinet and is generally a 0- or 1-based index indicating the ordinal of the position of the cabinet within a set.

See Also

SPFILENOTIFY_NEEDNEWCABINET, **FILE_IN_CABINET_INFO**

FILE_IN_CABINET_INFO Quick Info

Overview

Overview

[New - Windows NT]

The **FILE_IN_CABINET_INFO** structure provides information about a file found in the cabinet. The **SetupIterateCabinet** function sends this structure as one of the parameters when it sends a SPFILENOTIFY_FILEINCABINET notification to the cabinet callback routine.

```
typedef struct _FILE_IN_CABINET_INFO {
    PCTSTR NameInCabinet;
    DWORD  FileSize;
    DWORD  Win32Error;
    WORD   DosDate;
    WORD   DosTime;
    WORD   DosAttribs;
    TCHAR  FullTargetName[MAX_PATH];
} FILE_IN_CABINET_INFO, *PFILE_IN_CABINET_INFO;
```

Members

NameInCabinet
Specifies the filename as it exists within the cabinet file.

FileSize
Specifies the uncompressed size of the file in the cabinet.

Win32Error
If applicable, the Win32 error value associated with the file in the cabinet.

DosDate
The date that the file was last saved.

DosTime
The MS-DOS timestamp of the file in the cabinet.

DosAttribs
The attributes of the file in the cabinet.

FullTargetName[MAX_PATH]
The target path and filename.

See Also

CABINET_INFO

FILEPATHS Quick Info

Overview

Overview

[New - Windows NT]

The **FILEPATHS** structure stores source and target path information. The setup functions send the **FILEPATHS** structure as a parameter in several of the notifications sent to callback routines. For more information, see Notifications.

```
typedef struct _FILEPATHS {
    PCTSTR Target;
    PCTSTR Source;
    UINT   Win32Error;
    DWORD  Flags;
} FILEPATHS, *PFILEPATHS;
```

Members

Target

The path to the target file.

Source

The path to the source file. This member is not used when the **FILEPATHS** structure is used with a file delete operation.

Win32Error

If an error occurs, this parameter takes the Win32 error value associated with the specified paths. If no error has occurred, it takes the value NO_ERROR.

Flags

Specifies additional information that depends on the notification sent with the **FILEPATHS** structure.

For SPFILENOTIFY_COPYERROR notifications, **Flags** specifies dialog box behavior and can be one of the following values.

SP_COPY_WARNIFSKIP

Inform the user that skipping the file may affect the installation.

SP_COPY_NOSKIP

Do not offer the user the option to skip the file.

SP_COPY_NOBROWSE

Do not offer the user the option to browse.

For SPFILENOTIFY_FILEOPDELAYED notifications, **Flags** specifies the type of file operation delayed and can be one of the following values.

FILEOP_DELETE	A file delete operation was delayed.
FILEOP_COPY	A file copy operation was delayed.

INFCONTEXT Quick Info

Overview

Overview

[New - Windows NT]

The **INFCONTEXT** structure stores context information that functions such as SetupGetLineText use to navigate INF files.

```
typedef struct _INFCONTEXT {
    HINF Inf;                  //handle of the open INF file
    HINF CurrentInf;           //handle of the current INF file
    UINT Section;              //a section of the current INF file
    UINT Line;                 //a line of the INF section
} INFCONTEXT, *PINFCONTEXT;
```

Members

Inf

A handle to the INF file returned by SetupOpenInfFile.

CurrentInf

A pointer to the current INF file. The **Inf** member may point to multiple files if they were appended to the open INF file using SetupOpenAppendInfFile.

Section

Specifies a section in the current INF file.

Line

Specifies a line of the current section in the INF file.

Important The setup functions use this structure internally and it must not be accessed or modified by applications. It is included here for informational purposes only.

See Also

SetupFindFirstLine, SetupFindNextLine, SetupFindNextMatchLine

SOURCE_MEDIA Group

Group

Group

[New - Windows NT]

The **SOURCE_MEDIA** structure is used with the SPFILENOTIFY_NEEDMEDIA notification to pass source media information.

```
typedef struct _SOURCE_MEDIA {
    PCTSTR Reserved;
    PCTSTR Tagfile;
    PCTSTR Description;
    PCTSTR SourcePath;
    PCTSTR SourceFile;
    DWORD  Flags;
} SOURCE_MEDIA, *PSOURCE_MEDIA;
```

Members

Reserved

This member is not currently used.

Tagfile

This optional member specifies the tag file that can be used to identify the source media.

Description

The human-readable description of the source media.

SourcePath

The path to the source that needs the new media.

SourceFile

The source file to be retrieved from the new media.

Flags

Specifies copy style information that modifies how errors are handled. This member can be one or more of the following values:

SP_COPY_WARNIFSKIP

Inform the user that skipping the file may affect the installation.

SP_COPY_NOSKIP

Do not offer the user the option to skip the file.

SP_FLAG_CABINETCONTINUATION

The current source file is continued in another cabinet file.

SP_COPY_NOBROWSE

Do not offer the user the option to browse.

See Also

SPFILENOTIFY_NEEDMEDIA

SP_INF_INFORMATION Group

Group

Group

[New - Windows NT]

The **SP_INF_INFORMATION** structure stores information about an INF file, including the style, number of constituent INF files, and version data.

```
typedef struct _SP_INF_INFORMATION {
    DWORD InfStyle;            //the style of the INF file
    DWORD InfCount;            //number of constituent INF files
    BYTE  VersionData[ANYSIZE_ARRAY];
                               //array to store the INF information
} SP_INF_INFORMATION, *PSP_INF_INFORMATION;
```

Members

InfStyle

Specifies the style of the INF file. This member can be one of the following values.

Value	**Meaning**
INF_STYLE_NONE	Specifies that the style of the INF file is unrecognized or nonexistent.
INF_STYLE_OLDNT	Specifies a Windows NT 3.*x* style INF file.
INF_STYLE_WIN4	Specifies a Windows 95- or Windows NT-style INF file.

InfCount

Specifies the number of constituent INF files.

VersionData[ANYSIZE_ARRAY]

Stores information from the **Version** section of an INF file in an array of *ANYSIZE_ARRAY* bytes.

See Also

SetupGetInfInformation, **SetupQueryInfFileInformation**, **SetupQueryInfVersionInformation**

Functions

The following functions are used in the Setup API.

SetupAddToSourceList
SetupCancelTemporarySourceList
SetupCloseFileQueue
SetupCloseInfFile
SetupCommitFileQueue
SetupCopyError
SetupDecompressOrCopyFile
SetupDefaultQueueCallback
SetupDeleteError
SetupFindFirstLine
SetupFindNextLine
SetupFindNextMatchLine
SetupFreeSourceList
SetupGetBinaryField
SetupGetFieldCount
SetupGetFileCompressionInfo
SetupGetInfFileList
SetupGetInfInformation
SetupGetIntField
SetupGetLineByIndex
SetupGetLineCount
SetupGetLineText
SetupGetMultiSzField
SetupGetSourceFileLocation
SetupGetSourceFileSize
SetupGetSourceInfo
SetupGetStringField
SetupGetTargetPath
SetupInitDefaultQueueCallback
SetupInitDefaultQueueCallbackEx
SetupInitializeFileLog
SetupInstallFile
SetupInstallFileEx
SetupInstallFilesFromInfSection
SetupInstallFromInfSection
SetupInstallServicesFromInfSection
SetupIterateCabinet
SetupLogFile
SetupOpenAppendInfFile
SetupOpenFileQueue
SetupOpenInfFile
SetupOpenMasterInf
SetupPromptForDisk
SetupPromptReboot
SetupQueryFileLog
SetupQueryInfFileInformation
SetupQueryInfVersionInformation
SetupQuerySourceList
SetupQueueCopy
SetupQueueCopySection
SetupQueueDefaultCopy

SetupQueueDelete
SetupQueueDeleteSection
SetupQueueRename
SetupQueueRenameSection
SetupRemoveFileLogEntry
SetupRemoveFromSourceList
SetupRenameError
SetupScanFileQueue
SetupSetDirectoryId
SetupSetPlatformPathOverride
SetupSetSourceList
SetupTermDefaultQueueCallback
SetupTerminateFileLog

SetupAddToSourceList Group

Group

Group

[New - Windows NT]

The **SetupAddToSourceList** function appends a value to the list of installation sources for either the current user or the system. If the value already exists, it is removed first, so that duplicate entries are not created.

```
BOOL SetupAddToSourceList(
    DWORD Flags,        // specifies a list to append to
    PCTSTR Source       // the source to add to the list
  );
```

Parameters

Flags

Specifies which list to append the source to. This parameter can be any combination of the following values:

SRCLIST_SYSTEM

Add the source to the per-system list. The caller must be an administrator.

SRCLIST_USER

Add the source to the per-user list.

SRCLIST_SYSIFADMIN

If the caller is an administrator, the source is added to the per-system list; if the caller is not a member of the administrators local group, the source is added to the per-user list for the current user.

Note If a temporary list is currently in use (see **SetupSetSourceList**), the preceding flags are ignored and the source is added to the temporary list.

SRCLIST_APPEND

Add the source to the end of the list. If this flag is not specified, the source is added to the beginning of the list.

Source

Pointer to the source to add to the list.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

See Also

SetupRemoveFromSourceList, **SetupSetSourceList**

SetupCancelTemporarySourceList Group

Group

Group

[New - Windows NT]

The **SetupCancelTemporarySourceList** function cancels any temporary list and no-browse behavior and reestablishes standard list behavior.

```
BOOL SetupCancelTemporarySourceList(
    VOID      // takes no parameters
  );
```

Parameters

None.

Return Values

If a temporary list was in effect, the return value is TRUE. Otherwise, the return value is FALSE.

See Also

SetupSetSourceList

SetupCloseFileQueue Group

Group

Group

[New - Windows NT]

The **SetupCloseFileQueue** function destroys a setup file queue.

```
VOID SetupCloseFileQueue(
    HSPFILEQ QueueHandle      // handle to the file queue to close
  );
```

Parameters

QueueHandle

Handle to an open setup file queue.

Remarks

The **SetupCloseFileQueue** function does not flush the queue; pending operations are not performed. To commit a file queue before closing it call **SetupCommitFileQueue**.

See Also

SetupCommitFileQueue, **SetupInstallFile**, **SetupQueueCopy**, **SetupQueueDefaultCopy**, **SetupQueueDelete**, **SetupQueueRename**

SetupCloseInfFile Group

Group

Group

[New - Windows NT]

The **SetupCloseInfFile** function closes the INF file opened by a call to **SetupOpenInfFile** and any INF files appended to it by **SetupOpenAppendInfFile**.

```
VOID SetupCloseInfFile(
    HINF InfHandle       // handle to the INF file to close
  );
```

Parameters

InfHandle

Handle to the INF file to close.

See Also

SetupOpenInfFile, **SetupOpenAppendInfFile**

SetupCommitFileQueue Group

Group

Group

[New - Windows NT]

The **SetupCommitFileQueue** function performs file operations enqueued on a setup file queue.

```
BOOL SetupCommitFileQueue(
    HWND Owner,                      // optional; parent window
    HSPFILEQ QueueHandle,            // handle to the file queue
    PSP_FILE_CALLBACK MsgHandler,    // callback routine to use
    PVOID Context                    // passed to callback routine
  );
```

Parameters

Owner
This optional parameter supplies the handle of a window to use as the parent of any progress dialog boxes.

QueueHandle
Supplies a handle to a setup file queue, as returned by **SetupOpenFileQueue**.

MsgHandler
Supplies a callback routine to be notified of various significant events in the queue processing. For more information, see "Default Queue Callback Routine."

Context
Supplies a value that is passed to the callback function supplied by the *MsgHandler* parameter. If the default callback routine has been specified as *MsgHandler*, this context must be the context returned from **SetupInitDefaultQueueCallback** or **SetupInitDefaultQueueCallbackEx**.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Remarks

The callback routine specified in *MsgHandler* should be compatible with the the parameters that **SetupCommitFileQueue** passed to it during a queue commit.

If Unicode is defined in your callback application, and you specifiy *MsgHandler* as the default queue callback routine, the callback routine will expect Unicode parameters. Otherwise, the default queue callback routine will expect ANSI parameters.

See Also

SetupCloseFileQueue

SetupCopyError Group

Group

Group

[New - Windows NT]

The **SetupCopyError** function generates a dialog box to notify the user of a copy file error.

```
UINT SetupCopyError(
    HWND hwndParent,              // parent window for this dialog box
    PCTSTR DialogTitle,           // optional, title for this dialog box
    PCTSTR DiskName,              // optional, name of disk to insert
    PCTSTR PathToSource,          // failed source path
    PCTSTR SourceFile,            // source file of copy error
    PCTSTR TargetPathFile,        // optional, target file of copy error
    UINT Win32ErrorCode,          // error information
    DWORD Style,                  // dialog box formatting and display
    PTSTR PathBuffer,             // optional, receives new path info
    DWORD PathBufferSize,         // size of supplied buffer
    PDWORD PathRequiredSize       // optional, buffer size needed
  );
```

Parameters

hwndParent
Handle to the parent window for this dialog box.

DialogTitle
This optional parameter points to a null-terminated string specifying the dialog box title. If this parameter is NULL, the default title of "Copy Error" (localized to the system language) is used.

DiskName
This optional parameter points to a null-terminated string specifying the name of the disk to insert. If this parameter is NULL, the default name "(Unknown)" (localized to the system language) is used.

PathToSource
Pointer to the path component of the source file on which the operation failed; for example, F:\mips.

SourceFile
Pointer to a null-terminated string specifying the filename part of the file on which the operation failed. This filename is displayed if the user clicks on the **Details** or **Browse** buttons. The **SetupCopyError** function looks for the file using its compressed form names; therefore, you can pass *cmd.exe* and not worry that the file actually exists as *cmd.ex_* on the source media.

TargetPathFile
This optional parameter points to a null-terminated string that specifies the full path of the target file for rename and copy operations. If *TargetPathFile* is not specified, "(Unknown)" (localized to the system language) is used.

Win32ErrorCode
The Win32 error code encountered during the file operation. For information about Win32 error codes, see the WINERROR.H file included with the Win32 SDK.

Style
Specifies flags that control display formatting and behavior of the dialog box. This parameter can be one of the following flags:

IDF_NOBROWSE
Do not display the browse option.

IDF_NOSKIP

Do not display the skip file option.

IDF_NODETAILS

Do not display the details option. If this flag is set, the *TargetPathFile* and *Win32ErrorCode* parameters can be omitted.

IDF_NOCOMPRESSED

Do not check for compressed versions of the source file.

IDF_OEMDISK

The operation source is a disk provided by a hardware manufacturer.

IDF_NOBEEP

Prevent the dialog box from beeping to get the user's attention when it first appears.

IDF_NOFOREGROUND

Prevent the dialog box from becoming the foreground window.

IDF_WARNIFSKIP

Warn the user that skipping a file can affect the installation.

PathBuffer

This optional parameter points to a caller-supplied variable in which this function returns the path (not including the filename) of the location specified by the user through the dialog box.

PathBufferSize

Specifies the size of the buffer pointed to by *PathBuffer*. It should be at least MAX_PATH in length.

PathRequiredSize

This optional parameter points to a caller-supplied variable in which this function returns the required buffer size.

Return Values

The function returns one of the following values:

DPROMPT_SUCCESS

The requested disk/file is present and accessible. If *PathBuffer* was specified, it contains the path to the file (not including the filename).

DPROMPT_CANCEL

The user clicked on the **Cancel** button.

DPROMPT_SKIPFILE

The user clicked on the **Skip File** button.

DPROMPT_BUFFERTOOSMALL

The provided *PathBuffer* is too small. Check *PathRequiredSize* for the actual size needed.

DPROMPT_OUTOFMEMORY

There is insufficient memory to process the request.

Remarks

For the Unicode version of this function, the buffer sizes *ReturnBufferSize* and *RequiredSize* are specified in number of characters. This number includes the null terminator. For the ANSI version of this function, the sizes are specified in number of bytes.

If this function is called with a *ReturnBuffer* of NULL and a ReturnBufferSize of zero, the function puts the buffer size needed to hold the specified data into the variable pointed to by *RequiredSize*. If the function succeeds in this, the return value is NO_ERROR. Otherwise, the return value is one of the values specified preceding.

To avoid insufficient buffer errors, ReturnBuffer should be at least MAX_PATH.

See Also

SetupDeleteError, **SetupPromptForDisk**, **SetupRenameError**

SetupDecompressOrCopyFile Group

Group

Group

[New - Windows NT]

The **SetupDecompressOrCopyFile** function copies a file, decompressing it if necessary.

```
DWORD SetupDecompressOrCopyFile(
    PCTSTR SourceFileName,      // filename of the source file
    PCTSTR TargetFileName,      // filename after copy operation
    PUINT CompressionType       // optional, source file compression
  );
```

Parameters

SourceFileName

Filename of the file to copy. If *CompressionType* is not specified and the **SetupDecompressOrCopyFile** function does not find the file specified in *SourceFileName*, the function searches for the file with up to two alternate, "compressed-form" names. For example, if the file is F:\mips\cmd.exe and it is not found, the function searches for F:\mips\cmd.ex_ and, if that is not found, F:\mips\cmd.ex$ is searched for. If *CompressionType* is specified, no additional processing is performed on the filename; the file must exist exactly as specified or the function fails.

TargetFileName

Supplies the exact name of the target file that will be created by decompressing or copying the source file.

CompressionType

This optional parameter points to the compression type used on the source file. You can determine the compression type by calling **SetupGetFileCompressionInfo**. If this value is FILE_COMPRESSION_NONE, the file is copied (not decompressed) regardless of any compression in use on the source. If *CompressionType* is not specified, this routine determines the compression type automatically.

Return Values

The **SetupDecompressOrCopyFile** function returns a Win32 error code that indicates the outcome of the operation. For more information about Win32 error codes, see the WINERROR.H header file included with the Win32 SDK.

See Also

SetupGetFileCompressionInfo

SetupDefaultQueueCallback Group

Group

Group

[New - Windows NT]

The **SetupDefaultQueueCallback** function is the default queue callback routine included with the Setup API. You can use it to process notifications sent by the **SetupCommitFileQueue** function.

```
UINT SetupDefaultQueueCallback(
    PVOID Context,          // context used by the default callback routine
    UINT Notification,      // queue notification
    UINT Param1,            // additional notification information
    UINT Param2             // additional notification information
  );
```

Parameters

Context

Supplies a pointer to the context initialized by the **SetupInitDefaultQueueCallback** or **SetupInitDefaultQueueCallbackEx** functions.

Notification

Supplies the notification of a queue action. This parameter can be one of the following values:

SPFILENOTIFY_STARTQUEUE	Started enqueued file operations.
SPFILENOTIFY_ENDQUEUE	Finished enqueued file operations.
SPFILENOTIFY_STARTSUBQUEUE	Started a copy, rename, or delete subqueue.
SPFILENOTIFY_ENDSUBQUEUE	Finished a copy, rename, or delete subqueue.
SPFILENOTIFY_STARTRENAME	Started a rename operation.
SPFILENOTIFY_ENDRENAME	Finished a rename operation.
SPFILENOTIFY_RENAMEERROR	Encountered an error while renaming a file.
SPFILENOTIFY_STARTDELETE	Started a delete operation.
SPFILENOTIFY_ENDDELETE	Finished a delete operation.
SPFILENOTIFY_DELETEERROR	Encountered an error while deleting a file.
SPFILENOTIFY_STARTCOPY	Started a copy operation.
SPFILENOTIFY_ENDCOPY	Finished a copy operation.
SPFILENOTIFY_COPYERROR	Encountered an error while copying a file.
SPFILENOTIFY_NEEDMEDIA	New media is required.
SPFILENOTIFY_LANGMISMATCH	Existing target file is in a different language than the source.
SPFILENOTIFY_TARGETEXISTS	Target file exists.
SPFILENOTIFY_TARGETNEWER	Existing target file is newer than source.

Param1

Specifies additional message information. The content of this parameter depends on the value of the *Notification* parameter.

Param2

Specifies additional message information. The content of this parameter depends on the value of the *Notification* parameter.

Return Values

Returns an unsigned integer to **SetupCommitFileQueue** that can be the following values.

File Directive	Description
FILEOP_ABORT	Abort the operation.
FILEOP_DOIT	Perform the file operation.
FILEOP_SKIP	Skip the operation.
FILEOP_RETRY	Retry the operation.
FILEOP_NEWPATH	Get a new path for the operation.

Remarks

The **SetupDefaultQueueCallback** function is usually only called explicitly by a custom queue callback routine. The custom callback handles a subset of the queue commit notifications and calls the **SetupDefaultQueueCallback** function to handle the rest of the notifications.

For more information see, Queue Notifications.

See Also

SetupCommitFileQueue

SetupDeleteError Group

Group

Group

[New - Windows NT]

The **SetupDeleteError** function generates a dialog box that informs the user of a delete error.

```
UINT SetupDeleteError(
    HWND hwndParent,          // parent window for this dialog box
    PCTSTR DialogTitle,       // optional, title for this dialog box
    PCTSTR File,              // file that caused the delete error
    UINT Win32ErrorCode,      // specifies the error that occurred
    DWORD Style               // specifies formatting for the dialog box
  );
```

Parameters

hwndParent

Handle to the parent window for this dialog box.

DialogTitle

This optional parameter points to a null-terminated string specifying the error dialog box title. If this parameter is NULL, the default title of "Delete Error" (localized) is used.

File

Pointer to a null-terminated string specifying the full path of the file on which the delete operation failed.

Win32ErrorCode

The Win32 error code encountered during the file operation.

Style

Specifies flags that control display formatting and behavior of the dialog box. This parameter can be one of the following flags:

IDF_NOBEEP

Prevent the dialog box from beeping to get the user's attention when it first appears.

IDF_NOFOREGROUND

Prevent the dialog box from becoming the foreground window.

Return Values

This function returns one of the following values:

DPROMPT_SUCCESS

The user retried the operation and it was successful.

DPROMPT_CANCEL

The user clicked the **Cancel** button.

DPROMPT_SKIPFILE

The user clicked the **Skip File** button.

DPROMPT_OUTOFMEMORY

There is insufficient memory to process the request.

See Also

SetupCopyError, **SetupPromptForDisk**, **SetupRenameError**

SetupFindFirstLine Group

Group

Group

[New - Windows NT]

The **SetupFindFirstLine** function locates a line in the specified section of an INF file. If the *Key* parameter is NULL, **SetupFindFirstLine** returns the first line of the section.

```
BOOL SetupFindFirstLine(
    HINF InfHandle,              // handle to an INF file
    PCTSTR Section,              // section in which to find a line
    PCTSTR Key,                  // optional, key to search for
    PINFCONTEXT Context          // context of the found line
  );
```

Parameters

InfHandle

Handle to the INF file to query.

Section

Pointer to a null-terminated string specifying the section of the INF file(s) to search in.

Key

This optional parameter points to a null-terminated string specifying the key to search for within the section. If *Key* is NULL, the first line in the section is returned.

Context

Pointer to a structure in which this function returns the context information used internally by the INF handle. Applications must not overwrite values in this structure.

Return Values

If the function could not find a line, the return value is FALSE.

Remarks

If the *InfHandle* parameter references multiple INF files that have been appended together using **SetupOpenAppendInfFile**, the **SetupFindFirstLine** function searches across the specified section in all of the files referenced by the specified HINF.

See Also

SetupFindNextLine, **SetupFindNextMatchLine**, **SetupGetLineByIndex**

SetupFindNextLine Group

Group

Group

[New - Windows NT]

The **SetupFindNextLine** returns the location of the next line in an INF file section relative to *ContextIn.Line*.

```
BOOL SetupFindNextLine(
    PINFCONTEXT ContextIn,       // starting context in an INF file
    PINFCONTEXT ContextOut       // context of the next line
  );
```

Parameters

ContextIn

Pointer to the INF file context retrieved by a call to the **SetupFindFirstLine** function.

ContextOut

Pointer to a caller-supplied variable in which this function returns the context of the found line. *ContextOut* can point to *ContextIn* if the caller wishes.

Return Values

If this function finds the next line, the return value is TRUE. Otherwise, the return value is FALSE.

Remarks

If *ContextIn.Inf* references multiple INF files that have been appended together using **SetupOpenAppendInfFile**, this function searches across the specified section in all files referenced by the HINF to locate the next line.

See Also

SetupFindFirstLine, **SetupFindNextMatchLine**, **SetupGetLineByIndex**

SetupFindNextMatchLine Group

Group

Group

[New - Windows NT]

The **SetupFindNextMatchLine** function returns the location of the next line in an INF file relative to *ContextIn.Line* that matches a specified key.

```
BOOL SetupFindNextMatchLine(
    PINFCONTEXT ContextIn,      // starting context in an INF file
    PCTSTR Key,                 // optional, key to match
    PINFCONTEXT ContextOut      // context of the the found line
  );
```

Parameters

ContextIn
Specifies a pointer to an INF file context, as retrieved by a call to the **SetupFindFirstLine** function.

Key
If this optional parameter is specified, it supplies a key to match. If *Key* is not specified, the **SetupFindNextMatchLine** function is equivalent to the **SetupFindNextLine** function.

ContextOut
Pointer to a caller-supplied variable in which this function returns the context of the found line. *ContextOut* can point to *ContextIn* if the caller wishes.

Return Values

The function returns TRUE if it finds a matching line. Otherwise, the return value is FALSE.

Remarks

If *ContextIn.Inf* references multiple INF files that have been appended together using **SetupOpenAppendInfFile**, the **SetupFindNextMatchLine** function searches across the specified section in all files referenced by the HINF to locate the next matching line.

See Also

SetupFindFirstLine, **SetupFindNextLine**, **SetupGetLineByIndex**

SetupFreeSourceList Group

Group

Group

[New - Windows NT]

The **SetupFreeSourceList** function frees the system resources allocated to a source list.

```
BOOL SetupFreeSourceList(
    PCTSTR **List,      // an array of sources to free
    UINT Count          // the number of sources in the array
  );
```

Parameters

List

Specifies a pointer to an array of sources from **SetupQuerySourceList**. When the function returns, this pointer is set to NULL.

Count

Specifies the number of sources in the list.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

See Also

SetupCancelTemporarySourceList, **SetupSetSourceList**

SetupGetBinaryField Group

Group

Group

[New - Windows NT]

The **SetupGetBinaryField** function retrieves binary data from a line in an INF file section, from the specified field to the end of the line.

```
BOOL SetupGetBinaryField(
    PINFCONTEXT Context,        // context of a line in an INF file
    DWORD FieldIndex,           // index of the starting field
    BYTE * ReturnBuffer,        // optional, receives the fields
    DWORD ReturnBufferSize,     // size of the supplied buffer
    LPDWORD RequiredSize        // optional, buffer size needed
  );
```

Parameters

Context

Supplies INF context for the line.

FieldIndex

The 1-based index of the starting field within the specified line from which the binary data should be retrieved. The binary data is built from each field, starting at this point to the end of the line. Each field corresponds to 1 byte and is in hexadecimal notation. A *FieldIndex* of 0 is not valid with this function.

ReturnBuffer

This optional parameter points to a caller-supplied buffer that receives the binary data.

ReturnBufferSize

Specifies the size of the buffer pointed to by *ReturnBuffer*.

RequiredSize

This optional parameter points to a caller-supplied variable that receives the required size for the buffer pointed to *ReturnBuffer*. If the size needed is larger than the value specified by *ReturnBufferSize*, the function fails and a call to **GetLastError** returns ERROR_INSUFFICIENT_BUFFER.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

GetLastError returns ERROR_INVALID_DATA if a field that **SetupGetBinaryField** retrieves is not a valid hexadecimal number in the range 0-FF.

Remarks

To better understand how this function works, consider the following line from an INF file.

```
X=34,FF,00,13
```

If **SetupGetBinaryField** was called on the preceding line, the binary values 34, FF, 00, and 13 would be put into the buffer specified by *ReturnBuffer*.

For the Unicode version of this function, the buffer sizes *ReturnBufferSize* and *RequiredSize* are specified in number of characters. This number includes the null terminator. For the ANSI version of this function, the sizes are specified in number of bytes.

If this function is called with a *ReturnBuffer* of NULL and a ReturnBufferSize of zero, the function puts the buffer size needed to hold the specified data into the variable pointed to by *RequiredSize*. If the function succeeds in this, the return value is TRUE. Otherwise, the return value is FALSE and extended error information can be obtained by calling **GetLastError**.

Thus, you can call the function once to get the required buffer size, allocate the necessary memory, and then call the function a second time to retrieve the data. Using this technique, you can avoid errors due to an insufficient buffer size.

See Also

SetupGetIntField, **SetupGetMultiSzField**, **SetupGetStringField**

SetupGetFieldCount Group

Group

Group

[New - Windows NT]

The **SetupGetFieldCount** function retrieves the number of fields in the specified line in an INF file.

```
DWORD SetupGetFieldCount(
    PINFCONTEXT Context       //specifies the line to count fields in
  );
```

Parameters

Context

Pointer to the context for a line in an INF file.

Return Values

This function returns the number of fields on the line. If *Context* is invalid, 0 is returned.

See Also

SetupGetLineCount

SetupGetFileCompressionInfo Group

Group

Group

[New - Windows NT]

The **SetupGetFileCompressionInfo** function examines a physical file to determine if it is compressed and gets its full path, size, and the size of the uncompressed target file.

```
DWORD SetupGetFileCompressionInfo(
    PCTSTR SourceFileName,            // file to investigate
    PTSTR *ActualSourceFileName,      // receives compressed name
    PDWORD SourceFileSize,            // receives compressed size
    PDWORD TargetFileSize,            // receives uncompressed size
    PUINT CompressionType             // receives compression type
  );
```

Parameters

SourceFileName

Filename of the file about which information is required. If the file is not found on the source media exactly as named, the file is searched for with up to two alternate "compressed-form" names. For example, if the file is F:*\mips\cmd.exe* and it is not found, F:*\mpis\cmd.ex_* is searched for and, if that is not found, F:*\mips\cmd.ex$* is searched for.

ActualSourceFileName

Supplies a pointer to a caller-supplied variable in which this function returns the full path of the file that was actually located. The caller can free the pointer with a call to **LocalFree**. The path is valid only if the function returns NO_ERROR.

SourceFileSize

Supplies a pointer to a caller-supplied variable in which this function returns the size of the file in its current form which is the current size of the filenamed by *ActualSourceFileName*. The size is determined by examining the source file; it is not retrieved from an INF file. The source file size is valid only if the function returns NO_ERROR.

TargetFileSize

Supplies a pointer to a caller-supplied variable in which this function returns the size the file will occupy when it is uncompressed or copied. If the file is not compressed, this value will be the same as *SourceFileSize*. The size is determined by examining the file; it is not retrieved from an INF file. The target file size is valid only if the function returns NO_ERROR.

CompressionType

Supplies a pointer to a caller-supplied variable in which this function returns a value indicating the type of compression used on *ActualSourceFileName*. The compression type is valid only if the function returns NO_ERROR. The value can be one of the following flags:

FILE_COMPRESSION_NONE

The source file is not compressed with a recognized compression algorithm.

FILE_COMPRESSION_WINLZA

The source file is compressed with winlza (using *compress.exe* without the -z switch or LZXxx Win32 functions).

FILE_COMPRESSION_MSZIP

The source file is compressed with mszip (using *compress.exe* with the -z switch).

Return Values

The function returns a Win32 error code that indicates the outcome of the file search. The error code can be one of the following:

ERROR_FILE_NOT_FOUND
: The file cannot be found.

NO_ERROR
: The file was located and the output parameters were filled in.

Remarks

Because **SetupGetFileCompressionInfo** determines the compression by referencing the physical file, your setup application should ensure that the file is present before calling **SetupGetFileCompressionInfo**.

See Also

SetupDecompressOrCopyFile

SetupGetInfFileList Group

Group

Group

[New - Windows NT]

The **SetupGetInfFileList** function returns a list of INF files located in a caller-specified directory to a call-supplied buffer.

```
BOOL SetupGetInfFileList(
    PCTSTR DirectoryPath,       // optional, the directory path
    DWORD InfStyle,             // style of the INF file
    PTSTR ReturnBuffer,         // optional, receives the file list
    DWORD ReturnBufferSize,     // size of the supplied buffer
    PDWORD RequiredSize         // optional, buffer size needed
  );
```

Parameters

DirectoryPath

An optional parameter that points to a null-terminated string containing the path of the directory in which to search. If this value is NULL, the *%windir%\inf* directory is used.

InfStyle

Specifies the style of INF file to search for. May be a combination of the following flags.

INF_STYLE_OLDNT

Windows NT 3.x script-based INF files.

INF_STYLE_WIN4

Windows 95- or Windows NT 4.0-style INF files.

ReturnBuffer

If not NULL, points to a caller-supplied buffer in which this function returns the list of all INF files of the desired style(s) that were found in the specified subdirectory. Filenames are null-terminated, with an extra null at the end of the list. The filenames do not include the path.

ReturnBufferSize

Specifies the size of the buffer pointed to by the *ReturnBuffer* parameter. If *ReturnBuffer* is not specified, *ReturnBufferSize* is ignored.

RequiredSize

If not NULL, points to a caller-supplied variable in which this function returns the required size for the buffer pointed to by the *ReturnBuffer* parameter. If *ReturnBuffer* is specified and the size needed is larger than *ReturnBufferSize*, the function fails and a call to **GetLastError** returns ERROR_INSUFFICIENT_BUFFER.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Remarks

For the Unicode version of this function, the buffer sizes *ReturnBufferSize*, and *RequiredSize* are specified in number of characters. This number includes the null terminator. For the ANSI version of this function, the sizes are specified in number of bytes.

If this function is called with a *ReturnBuffer* of NULL and a ReturnBufferSize of zero, the function puts the buffer size needed to hold the specified data into the variable pointed to by *RequiredSize*. If the function succeeds in this, the return value is TRUE. Otherwise, the return value is FALSE and extended error information can be obtained by calling **GetLastError**.

Thus, you can call the function once to get the required buffer size, allocate the necessary memory, and then call the function a second time to retrieve the data. Using this technique, you can avoid errors due to an insufficient buffer size.

If multiple INF file styles are returned by this function, the style of a particular INF file can be determined by calling the **SetupGetInfInformation** function.

See Also

SetupGetInfInformation

SetupGetInfInformation Group

Group

Group

[New - Windows NT]

The **SetUpGetInfInformation** function returns the **SP_INF_INFORMATION** structure for the specified INF file to a caller-supplied buffer.

```
BOOL SetupGetInfInformation(
    LPCVOID InfSpec,                           // handle or filename of the INF file
    DWORD SearchControl,                       // how to search for the INF file
    PSP_INF_INFORMATION Return Buffer,         // optional, receives the INF info
    DWORD ReturnBufferSize,                    // size of the supplied buffer
    PDWORD RequiredSize                        // optional, buffer size needed
  );
```

Parameters

InfSpec

A handle or a filename for an INF file, depending on the value of *SearchControl*.

SearchControl

This parameter can be one of the following constants:

INFINFO_INF_SPEC_IS_HINF

InfSpec is an INF handle. A single INF handle may reference multiple INF files if they have been append-loaded together. If it does, the structure returned by this function contains multiple sets of information.

INFINFO_INF_NAME_IS_ABSOLUTE

The string specified for *InfSpec* is a full path. No further processing is performed on *InfSpec*.

INFINFO_DEFAULT_SEARCH

Search the default locations for the INF file specified for *InfSpec*, which is assumed to be a filename only. The default locations are *%windir%\inf*, followed by *%windir%\system32*.

INFINFO_REVERSE_DEFAULT_SEARCH

Same as INFINFO_DEFAULT_SEARCH, except the default locations are searched in reverse order.

INFINFO_INF_PATH_LIST_SEARCH

Search for the INF in each of the directories listed in the *DevicePath* value entry under:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion
```

ReturnBuffer

If not NULL, *ReturnBuffer* points to a caller-supplied buffer in which this function returns the **SP_INF_INFORMATION** structure.

ReturnBufferSize

Size of the *ReturnBuffer*.

RequiredSize

If not NULL, points to a caller-supplied variable in which this function returns the required size for the buffer pointed to by *ReturnBuffer*. If *ReturnBuffer* is specified and the size needed is larger than *ReturnBufferSize*, the function fails and a call to **GetLastError** returns ERROR_INSUFFICIENT_BUFFER.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

If the INF file could not be located, the function returns FALSE and a subsequent call to **GetLastError** returns ERROR_FILE_NOT_FOUND.

Remarks

For the Unicode version of this function, the buffer sizes *ReturnBufferSize* and *RequiredSize* are specified in number of characters. This number includes the null terminator. For the ANSI version of this function, the sizes are specified in number of bytes.

If this function is called with a *ReturnBuffer* of NULL and a ReturnBufferSize of zero, the function puts the buffer size needed to hold the specified data into the variable pointed to by *RequiredSize*. If the function succeeds in this, the return value is TRUE. Otherwise, the return value is FALSE and extended error information can be obtained by calling **GetLastError**.

Thus, you can call the function once to get the required buffer size, allocate the necessary memory, and then call the function a second time to retrieve the data. Using this technique, you can avoid errors due to an insufficient buffer size.

See Also

SetupQueryInfFileInformation, **SetupQueryInfVersionInformation**

SetupGetIntField Group

Group

Group

[New - Windows NT]

The **SetupGetIntField** function retrieves an integer value from the specified field of a line in an INF file.

```
BOOL SetupGetIntField(
    PINFCONTEXT Context,        // context of a line in an INF file
    DWORD FieldIndex,           // index of an integer field in the line
    PINT IntegerValue           // receives the retreived integer field
  );
```

Parameters

Context
Pointer to the context for a line in an INF file.

FieldIndex
The 1-based index of the field within the specified line from which the integer should be retrieved.

A *FieldIndex* of 0 can be used to retrieve an integer key (For example, consider the following INF line, 431 = 1, 2, 4. The value 431 would be put into the variable pointed at by *IntegerValue* if **SetupGetIntField** was called with a *FieldIndex* of 0).

Integer Value
Pointer to a caller-supplied variable that receives the integer. If the field is not an integer, the function fails and a call to **GetLastError** returns ERROR_INVALID_DATA.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Remarks

The integer field may start with a positive (+) or negative (-) sign. It will be interpreted as a decimal number, unless it is prefixed in the file with 0x or 0X, in which case it is hexadecimal.

See Also

SetupGetBinaryField, **SetupGetMultiSzField**, **SetupGetStringField**

SetupGetLineByIndex Group

Group

Group

[New - Windows NT]

The **SetupGetLineByIndex** function locates a line by its index value in the specified section in the INF file.

```
BOOL SetupGetLineByIndex(
    HINF InfHandle,           // handle to the INF file
    PCTSTR Section,           // section that contains the line
    DWORD Index,              // the index of the line to find
    PINFCONTEXT Context       // context that specifies the found line
  );
```

Parameters

InfHandle
  Handle of the INF file.

Section
  Pointer to a null-terminated string specifying the section of the INF file to search.

Index
  Specifies the index of the line to be located. The total number of lines in a particular section can be found with a call to **SetupGetLineCount**.

Context
  Points to a caller-supplied variable in which the function returns the context information for the found line.

Return Values

If the function succeeds, the return value is TRUE. If the function fails, the return value is FALSE.

Remarks

If *InfHandle* references multiple INF files that have been appended together using **SetupOpenAppendInfFile**, this function searches across the specified section in all files referenced by the HINF to locate the indexed line.

See Also

SetupFindFirstLine, **SetupFindNextLine**, **SetupFindNextMatchLine**

SetupGetLineCount Group

Group

Group

[New - Windows NT]

The **SetupGetLineCount** function returns the number of lines in a specified section of an INF file.

```
LONG SetupGetLineCount(
    HINF InfHandle,        // handle to the INF file
    PCTSTR Section         // the section in which to count lines
  );
```

Parameters

InfHandle

Handle of the INF file.

Section

Pointer to a null-terminated string that specifies the section in which you want to count the lines.

Return Values

If *InfHandle* references multiple INF files that have been appended together using **SetupOpenAppendInfFile**, this function returns the sum of the lines in all the INF files containing the specified section. A return value of 0 specifies an empty section. If the section does not exist, the function returns -1.

SetupGetLineText Group

Group

Group

[New - Windows NT]

The **SetupGetLineText** function returns the contents of a line in an INF file in a compact form. The line to retrieve can be specified by an **INFCONTEXT** structure returned from a **SetupFindLineXXX** function, or by explicitly passing in the INF handle, section, and key of the desired line.

```
BOOL SetupGetLineText(
    PINFCONTEXT Context,        // optional, context of an INF file
    HINF InfHandle,             // optional, handle to an INF file
    PCTSTR Section,             // optional, section in an INF file
    PCTSTR Key,                 // optional, key to look for
    PTSTR ReturnBuffer,         // optional, receives the line text
    DWORD ReturnBufferSize,     // size of the supplied buffer
    PDWORD RequiredSize         // optional, buffer size needed
  );
```

Parameters

Context

Supplies context for a line in an INF file whose text is to be retrieved. If *Context* is NULL, *InfHandle*, *Section*, and *Key* must be specified.

InfHandle

Handle of the INF file to query. This parameter is used only if *Context* is NULL.

Section

Pointer to a null-terminated string that specifies the section containing the key name of the line whose text is to be retrieved. This parameter is used only if *Context* is NULL.

Key

Pointer to a null-terminated string containing the key name whose associated string is to be retrieved. This parameter is used only if *Context* is NULL.

ReturnBuffer

If not NULL, points to a caller-supplied buffer in which this function returns the contents of the line.

ReturnBufferSize

Specifies the size of the buffer pointed to by the *ReturnBuffer* parameter.

RequiredSize

If not NULL, points to a caller-supplied variable in which this function returns the required size for the buffer pointed to by the *ReturnBuffer* parameter. If *ReturnBuffer* is specified and the size needed is larger than the value specified in the *ReturnBufferSize* parameter, the function fails and does not store data in the buffer.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Remarks

For the Unicode version of this function, the buffer sizes *ReturnBufferSize* and *RequiredSize* are specified in number of characters. This number includes the null terminator. For the ANSI version of this function, the sizes are specified in number of bytes.

If this function is called with a *ReturnBuffer* of NULL and a ReturnBufferSize of zero, the function puts the buffer size needed to hold the specified data into the variable pointed to by *RequiredSize*. If the function succeeds in this, the return value is TRUE. Otherwise, the return value is FALSE and extended error information can be obtained by calling **GetLastError**.

Thus, you can call the function once to get the required buffer size, allocate the necessary memory, and then call the function a second time to retrieve the data. Using this technique, you can avoid errors due to an insufficient buffer size.

This function returns the contents of a line in a compact format. All extraneous white space is removed and multi-line values are converted into a single contiguous string. For example, this line:

```
HKLM, , PointerClass0, 1 \
; This is a comment
01, 02, 03
```

would be returned as:

```
HKLM,,PointerClass0,1,01,02,03
```

See Also

SetupFindFirstLine, **SetupFindNextLine**, **SetupFindNextMatchLine**, **SetupGetLineByIndex**

SetupGetMultiSzField Group

Group

Group

[New - Windows NT]

The **SetupGetMultiSzField** function retrieves multiple strings stored in a line of an INF file, from the specified field to the end of the line.

```
BOOL SetupGetMultiSzField(
    PINFCONTEXT Context,         // context of a line in an INF file
    DWORD FieldIndex,            // index of the starting field
    PTSTR ReturnBuffer,          // optional, receives the fields
    DWORD ReturnBufferSize,      // size of the supplied buffer
    LPDWORD RequiredSize         // optional, buffer size needed
  );
```

Parameters

Context

Pointer to the context for a line in an INF file.

FieldIndex

The 1-based index of the starting field within the specified line from which the strings should be retrieved. The string list is built from each field starting at this point to the end of the line. A *FieldIndex* of 0 is not valid with this function.

ReturnBuffer

This optional parameter points to a caller-supplied character buffer that receives the strings. Each string is null-terminated, with an extra null at the end of the string list.

ReturnBufferSize

Specifies the size of the buffer pointed to by *ReturnBuffer*.

RequiredSize

This optional parameter points to a caller-supplied variable that receives the size required for the buffer pointed to by *ReturnBuffer*. If the size needed is larger than the value specified by *ReturnBufferSize*, the function fails and a call to **GetLastError** returns ERROR_INSUFFICIENT_BUFFER.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Remarks

For the Unicode version of this function, the buffer sizes *ReturnBufferSize* and *RequiredSize* are specified in number of characters. This number includes the null terminator. For the ANSI version of this function, the sizes are specified in number of bytes.

If this function is called with a *ReturnBuffer* of NULL and a ReturnBufferSize of zero, the function puts the buffer size needed to hold the specified data into the variable pointed to by *RequiredSize*. If the function succeeds in this, the return value is TRUE. Otherwise, the return value is FALSE and extended error information can be obtained by calling **GetLastError**.

Thus, you can call the function once to get the required buffer size, allocate the necessary memory, and then call the function a second time to retrieve the data. Using this technique, you can avoid errors due to an insufficient buffer size.

See Also

SetupGetBinaryField, **SetupGetIntField**, **SetupGetStringField**

SetupGetSourceFileLocation Group

Group

Group

[New - Windows NT]

The **SetupGetSourceFileLocation** function retrieves the location of a source file listed in an INF file.

```
BOOL SetupGetSourceFileLocation(
    HINF InfHandle,                 // handle of an INF file
    PINFCONTEXT InfContext,         // optional, context of an INF file
    PCTSTR FileName,                // optional, source file to locate
    PUINT SourceId,                 // receives the source media ID
    PTSTR ReturnBuffer,             // optional, receives the location
    DWORD ReturnBufferSize,         // size of the supplied buffer
    PDWORD Required Size            // optional, buffer size needed
  );
```

Parameters

InfHandle

Handle of the INF file that contains the **SourceDisksNames** and **SourceDisksFiles** sections. If platform-specific sections exist for the user's system (for example, **SourceDisksNames.mips** and **SourceDisksFiles.mips**), the platform-specific section will be used.

InfContext

This optional parameter points to the context of a line in a **Copy Files** section for which the full source path is to be retrieved. If this parameter is NULL, *FileName* is searched for in the **SourceDisksFiles** section of the INF file specified by *InfHandle*.

FileName

This optional parameter points to a null-terminated string containing the filename (no path) for which to return the full source location. Either this parameter or *InfContext* must be specified.

SourceId

Points to a caller-supplied variable that receives the source identifier of the media where the file is located from the **SourceDisksNames** section of the INF file.

ReturnBuffer

This optional parameter points to a caller-supplied buffer to receive the relative source path. The source path does not include the filename itself, nor does it include a drive letter/network share name. The path does not start or end with a backslash (\), so the empty string specifies the root directory.

ReturnBufferSize

Specifies the size of the buffer pointed to by *ReturnBuffer*.

RequiredSize

This optional parameter points to a caller-supplied variable that receives the required size for the buffer pointed to by the *ReturnBuffer* parameter. If the required size is larger than the value specified by *ReturnBufferSize*, the function fails and **GetLastError** returns ERROR_INSUFFICIENT_BUFFER.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Remarks

For the Unicode version of this function, the buffer sizes *ReturnBufferSize*, and *ReturnBufferSize* are specified in number of characters. This number includes the null terminator. For the ANSI version of this function, the sizes are specified in number of bytes.

If this function is called with a *ReturnBuffer* of NULL and a ReturnBufferSize of zero, the function puts the buffer size needed to hold the specified data into the variable pointed to by *RequiredSize*. If the function succeeds in this, the return value is TRUE. Otherwise, the return value is FALSE and extended error information can be obtained by calling **GetLastError**.

Thus, you can call the function once to get the required buffer size, allocate the necessary memory, and then call the function a second time to retrieve the data. Using this technique, you can avoid errors due to an insufficient buffer size.

See Also

SetupGetSourceInfo, **SetupGetSourceFileSize**

SetupGetSourceFileSize Group

Group

Group

[New - Windows NT]

The **SetupGetSourceFileSize** function reads the uncompressed size of a source file listed in an INF file.

```
BOOL SetupGetSourceFileSize(
    HINF InfHandle,                // handle of the INF file
    PINFCONTEXT InfContext,        // optional, context of the INF file
    PCTSTR FileName,               // optional, filename to find size of
    PCTSTR Section,                //optional, section in an INF file
    PDWORD FileSize,               // receives the size of the file
    UINT RoundingFactor            // optional, round to a muliple of this
  );
```

Parameters

InfHandle

Handle of the loaded INF file that contains the **SourceDisksNames** and **SourceDisksFiles** sections. If platform-specific sections exist for the user's system (for example, **SourceDisksNames.mips** and **SourceDisksFiles.mips**), the platform-specific section will be used.

InfContext

This optional parameter points to a context for a line in a **Copy Files** section for which the size is to be retrieved. If *InfContext* is NULL, the *FileName* parameter is used.

FileName

This optional parameter points to a null-terminated string containing the filename (no path) for which to return the size. If this parameter is NULL as well as *InfContext*, then the *Section* parameter is used.

Section

This optional parameter points to a null-terminated string containing the name of a **Copy Files** section. If this parameter is specified, the total size of all files listed in the section is computed.

FileSize

Pointer to a caller-supplied variable that receives the size, in bytes, of the specified file(s).

RoundingFactor

This optional parameter supplies a value for rounding file sizes. All file sizes are rounded up to a multiple of this number before being added to the total size. Rounding is useful for more exact determinations of the space that a file will occupy on a given volume, because it allows the caller to have file sizes rounded up to a multiple of the cluster size. Rounding does not occur unless *RoundingFactor* is specified.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Remarks

One and only one of the optional parameters, *InfContext*, *FileName*, and *Section*, must be specified.

See Also

SetupGetSourceFileLocation

SetupGetSourceInfo Group

Group

Group

[New - Windows NT]

The **SetupGetSourceInfo** function retrieves the path, tag file, or media description for a source listed in an INF file.

```
BOOL SetupGetSourceInfo(
    HINF InfHandle,             // handle to the INF file
    UINT SourceId,              // ID of the source media
    UINT InfoDesired,           // information to retrieve
    PTSTR ReturnBuffer,         // optional, receives the info
    DWORD ReturnBufferSize,     // size of the supplied buffer
    LPDWORD RequiredSize        // optional, buffer size needed
  );
```

Parameters

InfHandle

Handle of an open INF file that contains a **SourceDisksNames** section. If platform-specific sections exist for the user's system (for example, **SourceDisksNames.mips**), the platform-specific section will be used.

SourceId

Identifier for a source media. This value is used to search by key in the **SourceDisksNames** section.

InfoDesired

A value indicating what information is desired. Only one value may be specified per function call, and they cannot be combined. The following types of information can be retrieved from a **SourceDisksNames** section:

SRCINFO_PATH

The path specified for the source. This is not a full path, but the path relative to the installation root.

SRCINFO_TAGFILE

The tag file that identifies the source media, or if cabinets are used, the name of the cabinet file.

SRCINFO_DESCRIPTION

A description for the media.

ReturnBuffer

This optional parameter points to a caller-supplied buffer to receive the retrieved information. Path returns are guaranteed not to end with \.

ReturnBufferSize

Specifies the size of the buffer pointed to by *ReturnBuffer*.

RequiredSize

This optional parameter points to a caller-supplied variable that receives the required size for the buffer specified by *ReturnBuffer*. If *ReturnBuffer* is specified and the actual size needed is larger than the value specified by *ReturnBufferSize*, the function fails and a call to **GetLastError** returns ERROR_INSUFFICIENT_BUFFER.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Remarks

For the Unicode version of this function, the buffer sizes *ReturnBufferSize* and *RequiredSize* are specified in number of characters. This number includes the null terminator. For the ANSI version of this function, the sizes are specified in number of bytes.

If this function is called with a *ReturnBuffer* of NULL and a ReturnBufferSize of zero, the function puts the buffer size needed to hold the specified data into the variable pointed to by *RequiredSize*. If the function succeeds in this, the return value is TRUE. Otherwise, the return value is FALSE and extended error information can be obtained by calling **GetLastError**.

Thus, you can call the function once to get the required buffer size, allocate the necessary memory, and then call the function a second time to retrieve the data. Using this technique, you can avoid errors due to an insufficient buffer size.

See Also

SetupGetSourceFileLocation, **SetupGetSourceFileSize**, **SetupGetTargetPath**

SetupGetStringField Group

Group

Group

[New - Windows NT]

The **SetupGetStringField** function retrieves a string from the specified field of a line in an INF file.

```
BOOL SetupGetStringField(
    PINFCONTEXT Context,        // context of the INF file
    DWORD FieldIndex,           // index of the field to get
    PTSTR ReturnBuffer,         // optional, receives the field
    DWORD ReturnBufferSize,     // size of the provided buffer
    PDWORD RequiredSize         // optional, buffer size needed
  );
```

Parameters

Context

Pointer to the context for a line in an INF file.

FieldIndex

The 1-based index of the field within the specified line from which the string should be retrieved. Use a *FieldIndex* of 0 to retrieve a string key, if present.

ReturnBuffer

This optional parameter points to a caller-supplied buffer to which this function returns the string.

ReturnBufferSize

The size of the buffer pointed to by *ReturnBuffer*.

RequiredSize

This optional parameter points to a caller-supplied variable to which **SetupGetStringField** returns the required size for the buffer pointed to by the *ReturnBuffer* parameter. If *ReturnBuffer* is specified and the actual size needed is larger than the value specified by *ReturnBufferSize*, the function fails and does not store the string in the buffer. In this case, a call to **GetLastError** returns ERROR_INSUFFICIENT_BUFFER.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Remarks

For the Unicode version of this function, the buffer sizes *ReturnBufferSize*, and *ReturnBufferSize* are specified in number of characters. This number includes the null terminator. For the ANSI version of this function, the sizes are specified in number of bytes.

If this function is called with a *ReturnBuffer* of NULL and a ReturnBufferSize of zero, the function puts the buffer size needed to hold the specified data into the variable pointed to by *RequiredSize*. If the function succeeds in this, the return value is TRUE. Otherwise, the return value is FALSE and extended error information can be obtained by calling **GetLastError**.

Thus, you can call the function once to get the required buffer size, allocate the necessary memory, and then call the function a second time to retrieve the data. Using this technique, you can avoid errors due to an insufficient buffer size.

See Also

SetupGetIntField, **SetupGetBinaryField**, **SetupGetMultiSzField**

SetupGetTargetPath Group

Group

Group

[New - Windows NT]

The **SetupGetTargetPath** function determines the target directory for a file list section. The file list section can be a **Copy Files** section, a **Delete Files** section, or a **Rename Files** section. All the files in the section must be in a single directory that is listed in a **DestinationDirs** section of the INF file.

```
BOOL SetupGetTargetPath(
    HINF InfHandle,                // handle of the INF file
    PINFCONTEXT InfContext,        // optional, context of the INF file
    PCTSTR Section,                // optional, section in the INF file
    PTSTR ReturnBuffer,            // optional, receives the path info
    DWORD ReturnBufferSize,        // size of the supplied buffer
    PDWORD RequiredSize            // optional, buffer size needed
  );
```

Parameters

InfHandle
Handle of the load INF file that contains a **DestinationDirs** section.

InfContext
This optional parameter points to a INF context that specifies a line in a file list section whose destination directory is to be retrieved. If *InfContext* is NULL, then the *Section* parameter is used.

Section
This optional parameter specifies the name of a section of the INF file whose handle is *InfHandle*. **SetupGetTargetPath** retrieves the target directory for this section. The *Section* parameter is ignored if *InfContext* is specified. If neither *InfContext* nor *Section* is specified, the function retrieves the default target path from the INF file.

ReturnBuffer
This optional parameter points to a caller-supplied buffer to receive the fully qualified Win32 target path. The path is guaranteed not to end with \.

ReturnBufferSize
Specifies the size of the buffer pointed to by *ReturnBuffer*.

RequiredSize
This optional parameter points to a caller-supplied variable that receives the required size for the buffer pointed to by *ReturnBuffer*. If the actual size needed is larger than the value specified by *ReturnBufferSize*, the function fails and a call to **GetLastError** returns ERROR_INSUFFICIENT_BUFFER.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Remarks

For the Unicode version of this function, the buffer sizes *ReturnBufferSize* and *RequiredSize* are specified in number of characters. This number includes the null terminator. For the ANSI version of this function, the sizes are specified in number of bytes.

If this function is called with a *ReturnBuffer* of NULL and a ReturnBufferSize of zero, the function puts the buffer size needed to hold the specified data into the variable pointed to by *RequiredSize*. If the function succeeds in this, the return value is TRUE. Otherwise, the return value is FALSE and extended error information can be obtained by calling **GetLastError**.

Thus, you can call the function once to get the required buffer size, allocate the necessary memory, and then call the function a second time to retrieve the data. Using this technique, you can avoid errors due to an insufficient buffer size.

See Also

SetupGetSourceFileLocation, **SetupGetSourceInfo**, **SetupGetSourceFileSize**

SetupInitDefaultQueueCallback Group

Group

Group

[New - Windows NT]

The **SetupInitDefaultQueueCallback** function initializes the context used by the default queue callback routine included with the Setup API.

```
PVOID SetupInitDefaultQueueCallback(
    HWND OwnerWindow       // parent window of any dialog boxes created by the callback routine
  );
```

Parameters

OwnerWindow

Supplies the handle of the window to use as the parent of any dialog boxes generated by the default callback routine.

Return Values

A pointer to the context used by the default queue callback routine.

See Also

SetupDefaultQueueCallback, **SetupInitDefaultQueueCallbackEx**

SetupInitDefaultQueueCallbackEx Group

Group

Group

[New - Windows NT]

The **SetupInitDefaultQueueCallback** function initializes the context used by the default queue callback routine included with the Setup API in the same manner as **SetupInitDefaultQueueCallback**, except that an additional window is provided to the callback function to accept progress messages.

```
PVOID SetupInitDefaultQueueCallbackEx(
    HWND OwnerWindow,                // parent of callback dialog boxes
    HWND AlternateProgressWindow,    // handle to progress window
    UINT ProgressMessage,            // specifies progress message
    DWORD Reserved1,                 // wParam of a window message
    PVOID Reserved2                  // lParam of a window message
  );
```

Parameters

OwnerWindow
: Supplies the handle of the window to use as the parent of any dialog boxes generated by the default callback routine.

AlternateProgressWindow
: Supplies the handle to a window that will receive the progress messages.

ProgressMessage
: The message that is sent to *AlternateProgressWindow* once when the copy queue is started, and each time a file is copied.

Reserved1
: This parameter is the first message parameter that will be sent to the AlternateProgressWindow by the default callback routine.

Reserved2
: This parameter is the second message parameter that will be sent to the AlternateProgressWindow by the default callback routine.

Return Values

A pointer to the context used by the default queue callback routine.

Remarks

When the queue starts to commit the copy sub queue, the default queue callback routine will send a message to the window specified in *AlternateProgressWindow*. *Reserved1* will have the value 0, and *Reserved2* will contain a pointer to the number of enqueued file copy operations.

For each file copy operation completed, the default queue callback routine will send a message to *AlternateProgressWindow*, which can be used to 'tick' the progress bar. *Reserved1* will have the value 1, and *Reserved2* will be 0.

SetupInitDefaultQueueCallbackEx can be used to get the default behavior for disk prompting, error handling, and so on, and also provide a gauge embedded in a wizard page or other specialized dialog box.

See Also

SetupInitDefaultQueueCallback

SetupInitializeFileLog Group

Group

Group

[New - Windows NT]

The **SetupInitializeFileLog** function initializes a file to record installation operations and outcomes. This can be the system log, where the system tracks the files installed as part of Windows NT, or any other file.

```
HSPFILELOG SetupInitializeFileLog(
    PCTSTR LogFileName,      // optional, name of a log file to use
    DWORD Flags              // controls initialization
  );
```

Parameters

LogFileName

This optional parameter supplies the filename of the file to use as the log file. The *LogFileName* parameter must be specified if *Flags* does not include SPFILELOG_SYSTEMLOG. The *LogFileName* parameter must not be specified if *Flags* includes SPFILELOG_SYSTEMLOG.

Flags

A set of flags that controls the log file initialization. This parameter can be a combination of the following:

SPFILELOG_SYSTEMLOG

Use the Windows NT system file log which is used to track what files are installed as part of Windows NT. The user must be Administrator to specify this option unless SPFILELOG_QUERYONLY is specified and *LogFileName* is not specified. Do not specify SPFILELOG_SYSTEMLOG in combination with SPFILELOG_FORCENEW.

SPFILELOG_FORCENEW

If the log file exits, overwrite it. If the log file exists and this flag is not specified, any new files that are installed are added to the list in the existing log file. Do not specify in combination with SPFILELOG_SYSTEMLOG.

SPFILELOG_QUERYONLY

Open the log file for querying only.

Return Values

The function returns the handle to the log file if it is successful. Otherwise, the the return value is INVALID_HANDLE_VALUE and the logged error can be retrieved by a call to **GetLastError**.

See Also

SetupTerminateFileLog, **SetupLogFile**

SetupInstallFile Group

Group

Group

[New - Windows NT]

The **SetupInstallFile** function installs a file as specified either by an **INFCONTEXT** returned by **SetupFindXXXLine** or explicitly by the filename and path information.

```
BOOL SetupInstallFile(
    HINF InfHandle,                       // optional, handle to the INF file
    PINFCONTEXT InfContext,               // optional, context for a INF file line
    PCTSTR SourceFile,                    // optional, name of file to copy
    PCTSTR SourcePathRoot,                // optional, root path to source
    PCTSTR DestinationName,               // optional, filename after copy
    DWORD CopyStyle,                      // specifies copy behavior
    PSP_FILE_CALLBACK CopyMsgHandler,     // optional, callback routine to use
    PVOID Context                         // optional, callback routine context
  );
```

Parameters

InfHandle

This optional parameter points to the handle of an INF file that contains **SourceDisksNames** and **SourceDisksFiles** sections. If platform-specific sections exist for the user's system (for example, **SourceDisksNames.mips** and **SourceDisksFiles.mips**), the platform-specific section will be used. If *InfContext* is NULL and *CopySyle* includes SP_COPY_SOURCE_ABSOLUTE or SP_COPY_SOURCEPATH_ABSOLUTE, *InfHandle* is ignored.

InfContext

This optional parameter points to the context of a line in a **Copy Files** section in an INF file. The routine looks this file up in the **SourceDisksFiles** section of *InfHandle* to get file copy information. If *InfContext* is not specified, *SourceFile* must be.

SourceFile

This optional parameter supplies the filename (no path) of the file to copy. The file is looked up in the **SourceDisksFiles** section. The *SourceFile* parameter must be specified if *InfContext* is not, and is ignored if *InfContext* is specified.

SourcePathRoot

This optional parameter supplies the root path for the file to be copied (for example, A:\ or F:\). Paths in the **SourceDisksNames** section are appended to this path. The *SourcePathRoot* parameter is ignored if *CopyStyle* includes the SP_COPY_SOURCE_ABSOLUTE flag.

DestinationName

This optional parameter supplies the filename only (no path) of the target file. This parameter can be NULL to indicate that the target file should have the same name as the source file. If *InfContext* is not specified, *DestinationName* supplies the full target path and filename for the target.

CopyStyle

Flags that control the behavior of the file copy operation. These flags may be a combination of the following values:

SP_COPY_DELETESOURCE

Delete the source file upon successful copy. The caller is not notified if the delete fails.

SP_COPY_REPLACEONLY

Copy the file only if doing so would overwrite a file at the destination path. If the target doesn't exist, the function returns FALSE and **GetLastError** returns NO_ERROR.

SP_COPY_NEWER

Examine each file being copied to see if its version resources indicate that it is not newer than an existing copy on the target.

The file version information used during version checks is that specified in the **dwFileVersionMS** and **dwFileVersionLS** members of a **VS_FIXEDFILEINFO** structure, as filled in by the Win32 version functions. If one of the files does not have version resources, or if they have identical version information, the source file is considered newer.

If the source file is not newer, and *CopyMsgHandler* is specified, the caller is notified and may cancel the copy. If *CopyMsgHandler* is not specified, the file is not copied.

SP_COPY_NOOVERWRITE

Check whether the target file exists, and, if so, notify the caller who may veto the copy. If *CopyMsgHandler* is not specified, the file is not overwritten.

SP_COPY_NODECOMP

Do not decompress the file. When this flag is set, the target file is not given the uncompressed form of the source name (if appropriate). For example, copying f:*\mips\cmd.ex_* to *\\install\temp* results in a target file of *\\install\temp\cmd.ex_*. If the SP_COPY_NODECOMP flag was not specified, the file would be decompressed and the target would be called *\\install\temp\cmd.exe*. The filename part of *DestinationName*, if specified, is stripped and replaced with the filename of the source file. When SP_COPY_NODECOMP is specified, SP_COPY_LANGUAGEAWARE and SP_COPY_NEWER are ignored.

SP_COPY_LANGUAGEAWARE

Examine each file being copied to see if its language differs from the language of any existing file already on the target. If so, and *CopyMsgHandler* is specified, the caller is notified and may cancel the copy. If *CopyMsgHandler* is not specified, the file is not copied.

SP_COPY_SOURCE_ABSOLUTE

SourceFile is a full source path. Do not look it up in the **SourceDisksNames** section of the INF file.

SP_COPY_SOURCEPATH_ABSOLUTE

SourcePathRoot is the full path part of the source file. Ignore the relative source specified in the **SourceDisksNames** section of the INF file for the source media where the file is located. This flag is ignored if SP_COPY_SOURCE_ABSOLUTE is specified.

SP_COPY_FORCE_IN_USE

If the target exists, behave as if it is in use and queue the file for copying on the next system reboot.

SP_COPY_FORCE_NOOVERWRITE

Check whether the target file exists, and, if so, the file is not overwritten. The caller is not notified.

SP_COPY_FORCE_NEWER

Examine each file being copied to see if its version resources (or timestamps for non-image files) indicate that it is not newer than an existing copy on the target. If the file being copied is not newer, the file is not copied. The caller is not notified. The function returns FALSE, and **GetLastError** returns NO_ERROR.

CopyMsgHandler

This optional parameter points to a callback function to be notified of various conditions that may arise during the file copy.

Context

This optional parameter points to a caller-defined value that is passed as the first parameter of the callback function.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

If **GetLastError** returns NO_ERROR, the file copy operation was not completed because the callback function returned FALSE.

Remarks

If a UNC directory is specified as the target directory of a file installation, you must ensure it exists before you call **SetupInstallFile**. The setup functions do not check for the existence of and do not create UNC directories. If the target UNC directory does not exist, the file installation will fail.

See Also

SetupCloseFileQueue, **SetupCommitFileQueue**, **SetupOpenFileQueue**, **SetupQueueCopy**

SetupInstallFileEx Group

Group

Group

[New - Windows NT]

The **SetupInstallFileEx** function installs a file as specified either by an **INFCONTEXT** returned by **SetupFindXXXLine** or explicitly by the filename and path information. This function is the same as **SetupInstallFile**, except that a BOOL is returned indicating whether or not the file was in use.

```
BOOL SetupInstallFileEx(
    HINF InfHandle,                        // optional, handle to the INF file
    PINFCONTEXT InfContext,                // optional, context of an INF file
    PCTSTR SourceFile,                     // optional, filename of the source
    PCTSTR SourcePathRoot,                 // optional, root path to the source
    PCTSTR DestinationName,                // optional, filename after copy
    DWORD CopyStyle,                       // specifies copy behavior
    PSP_FILE_CALLBACK CopyMsgHandler,      // optional, callback routine to use
    PVOID Context,                         // optional, callback routine context
    PBOOL FileWasInUse                     // receives the file in use information
  );
```

Parameters

InfHandle

This optional parameter points to the handle of an INF file that contains the **SourceDisksNames** and **SourceDisksFiles** sections. If platform-specific sections exist for the user's system (for example, **SourceDisksNames.mips** and **SourceDisksFiles.mips**), the platform-specific section will be used. If *InfContext* is not specified and *CopySyle* includes SP_COPY_SOURCE_ABSOLUTE or SP_COPY_SOURCEPATH_ABSOLUTE, *InfHandle* is ignored.

InfContext

This optional parameter supplies context for a line in a **Copy Files** section in an INF file. The routine looks this file up in the **SourceDisksFiles** section of *InfHandle* to get file copy information. If *InfContext* is not specified, *SourceFile* must be.

SourceFile

This optional parameter supplies the filename (no path) of the file to copy. The file is looked up in the **SourceDisksFiles** section. The *SourceFile* parameter must be specified if *InfContext* is not. However, *SourceFile* is ignored if *InfContext* is specified.

SourcePathRoot

This optional parameter supplies the root path for the file to be copied (for example, A:\ or F:\). Paths in the **SourceDisksNames** section are appended to this path. The *SourcePathRoot* parameter is ignored if *CopyStyle* includes the SP_COPY_SOURCE_ABSOLUTE flag.

DestinationName

This optional parameter can be used to specify a new name for the copied file. If *InfContext* is specified, *DestinationName* supplies the filename only (no path) of the target file. This parameter can be NULL to indicate that the target file should have the same name as the source file. If *InfContext* is not specified, *DestinationName* supplies the full target path and filename for the target.

CopyStyle

Flags that control the behavior of the file copy operation. These flags may be a combination of the following values:

SP_COPY_DELETESOURCE

Delete the source file upon successful copy. The caller is not notified if the delete fails.

SP_COPY_REPLACEONLY

Copy the file only if doing so would overwrite a file at the destination path.

SP_COPY_NEWER

Examine each file being copied to see if its version resources indicate that it is not newer than an existing copy on the target.

The file version information used during version checks is that specified in the **dwFileVersionMS** and **dwFileVersionLS** members of a **VS_FIXEDFILEINFO** structure, as filled in by the Win32 version functions. If one of the files does not have version resources, or if they have identical version information, the source file is considered newer.

If the source file is not newer, and *CopyMsgHandler* is specified, the caller is notified and may cancel the copy. If *CopyMsgHandler* is not specified, the file is not copied.

SP_COPY_NOOVERWRITE

Check whether the target file exists, and, if so, notify the caller who may veto the copy. If *CopyMsgHandler* is not specified, the file is not overwritten.

SP_COPY_NODECOMP

Do not decompress the file. When this flag is set, the target file is not given the uncompressed form of the source name (if appropriate). For example, copying f:*\mips\cmd.ex_* to *\\install\temp* results in a target file of *\\install\temp\cmd.ex_*. If the SP_COPY_NODECOMP flag was not specified, the file would be decompressed and the target would be called *\\install\temp\cmd.exe*. The filename part of *DestinationName*, if specified, is stripped and replaced with the filename of the source file. When SP_COPY_NODECOMP is specified, SP_COPY_LANGUAGEAWARE and SP_COPY_NEWER are ignored.

SP_COPY_LANGUAGEAWARE

Examine each file being copied to see if its language differs from the language of any existing file already on the target. If so, and *CopyMsgHandler* is specified, the caller is notified and may cancel the copy. If *CopyMsgHandler* is not specified, the file is not copied.

SP_COPY_SOURCE_ABSOLUTE

SourceFile is a full source path. Do not look it up in the **SourceDisksNames** section of the INF file.

SP_COPY_SOURCEPATH_ABSOLUTE

SourcePathRoot is the full path part of the source file. Ignore the relative source specified in the **SourceDisksNames** section of the INF file for the source media where the file is located. This flag is ignored if SP_COPY_SOURCE_ABSOLUTE is specified.

SP_COPY_FORCE_IN_USE

If the target exists, behave as if it is in use and queue the file for copying on the next system reboot.

SP_COPY_IN_USE_NEEDS_REBOOT

If the file was in use during the copy operation, alert the user that the system needs to be rebooted.

SP_COPY_NO_SKIP

Do not give the user the option to skip a file.

SP_COPY_FORCE_NOOVERWRITE

Check whether the target file exists, and, if so, the file is not overwritten. The caller is not notified.

SP_COPY_FORCE_NEWER

Examine each file being copied to see if its version resources (or timestamps for non-image files) indicate that it is not newer than an existing copy on the target. If the file being copied is not newer, the file is not copied. The caller is not notified.

SP_COPY_WARNIFSKIP

If the user tries to skip a file, warn them that skipping a file may affect the installation. (Used for system-critical files.)

CopyMsgHandler

This optional parameter points to a callback function to be notified of various conditions that may arise during the file copy.

Context

Pointer to a caller-defined value that is passed as the first parameter of the callback function.

FileWasInUse

This optional parameter points to a caller-supplied variable in which this function returns a flag indicating whether the file was in use.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

If **GetLastError** returns NO_ERROR, the file copy operation was not completed because the callback function returned FALSE.

Remarks

This API is typically used when installing new versions of system files that are likely to be in use. It updates a BOOL value that indicates whether the file was in use. If the file was in use, then the file copy operation is post-poned until the system is rebooted.

If a UNC directory is specified as the target directory of a file installation, you must ensure it exists before you call **SetupInstallFileEx**. The setup functions do not check for the existence of and do not create UNC directories. If the target UNC directory does not exist, the file installation will fail.

See Also

SetupCloseFileQueue, **SetupCommitFileQueue**, **SetupInstallFile**, **SetupOpenFileQueue**, **SetupQueueCopy**, **SetupPromptReboot**

SetupInstallFilesFromInfSection Group

Group

Group

[New - Windows NT]

The **SetupInstallFilesFromInfSection** function queues all the files in an INF file **Copy Files** section to be installed.

```
BOOL SetupInstallFilesFromInfSection(
    HINF InfHandle,               // handle to the INF file
    HINF LayoutInfHandle,         // optional, layout INF handle
    HSPFILEQ FileQueue,           // handle to the file queue
    PCTSTR SectionName,           // section to install files from
    PCTSTR SourceRootPath,        // optional, root path to source files
    UINT CopyFlags                // optional, specifies copy behavior
  );
```

Parameters

InfHandle

Specifies a handle to an INF file that contains the section to be installed. The INF file must be a Windows 95- or Windows NT 4.0-style INF file; script-based INF files are not supported by **SetupInstallFilesFromInfSection**.

LayoutInfHandle

This optional parameter supplies a handle to the INF file that contains the **SourceDisksFiles** and **SourceDisksNames** sections. If *LayoutInfHandle* is not specified, then the **SourceDisksFiles** and **SourceDisksNames** sections from *InfHandle* will be used.

FileQueue

Supplies a handle to the queue to add installation operations to.

SectionName

Supplies the name of the section in *InfHandle* to operate on.

SourceRootPath

This optional parameter specifies a root path to the source files to copy. An example would be A:\ or \ *\pegasus\win\install*.

CopyFlags

This optional parameter specifies a set of flags that control the behavior of the file copy operation. These flags may be a combination of the following values:

SP_COPY_DELETESOURCE

Delete the source file upon successful copy. The caller is not notified if the delete fails.

SP_COPY_REPLACEONLY

Copy the file only if doing so would overwrite a file at the destination path.

SP_COPY_NEWER

Examine each file being copied to see if its version resources indicate that it is not newer than an existing copy on the target.

The file version information used during version checks is that specified in the **dwFileVersionMS** and **dwFileVersionLS** members of a **VS_FIXEDFILEINFO** structure, as filled in by the Win32 version functions. If one of the files does not have version resources, or if they have identical version information, the source file is considered newer.

If the source file is not newer, and *CopyMsgHandler* is specified, the caller is notified and may cancel the copy. If *CopyMsgHandler* is not specified, the file is not copied.

SP_COPY_NOOVERWRITE

Check whether the target file exists, and, if so, notify the caller who may veto the copy. If *CopyMsgHandler* is not specified, the file is not overwritten.

SP_COPY_NODECOMP
: Do not decompress the file. When this flag is set, the target file is not given the uncompressed form of the source name (if appropriate). For example, copying f:*\mips\cmd.ex_* to *\\install\temp* results in a target file of *\\install\temp\cmd.ex_*. If the SP_COPY_NODECOMP flag was not specified, the file would be decompressed and the target would be called *\\install\temp\cmd.exe*. The filename part of *DestinationName*, if specified, is stripped and replaced with the filename of the source file. When SP_COPY_NODECOMP is specified, SP_COPY_LANGUAGEAWARE and SP_COPY_NEWER are ignored.

SP_COPY_LANGUAGEAWARE
: Examine each file being copied to see if its language differs from the language of any existing file already on the target. If so, and *CopyMsgHandler* is specified, the caller is notified and may cancel the copy. If *CopyMsgHandler* is not specified, the file is not copied.

SP_COPY_SOURCE_ABSOLUTE
: *SourceFile* is a full source path. Do not look it up in the **SourceDisksNames** section of the INF file.

SP_COPY_SOURCEPATH_ABSOLUTE
: *SourcePathRoot* is the full path part of the source file. Ignore the relative source specified in the **SourceDisksNames** section of the INF file for the source media where the file is located. This flag is ignored if SP_COPY_SOURCE_ABSOLUTE is specified.

SP_COPY_FORCE_IN_USE
: If the target exists, behave as if it is in use and queue the file for copying on the next system reboot.

SP_COPY_IN_USE_NEEDS_REBOOT
: If the file was in use during the copy operation, alert the user that the system needs to be rebooted.

SP_COPY_NO_SKIP
: Do not give the user the option to skip a file.

SP_COPY_FORCE_NOOVERWRITE
: Check whether the target file exists, and, if so, the file is not overwritten. The caller is not notified.

SP_COPY_FORCE_NEWER
: Examine each file being copied to see if its version resources (or timestamps for non-image files) indicate that it is not newer than an existing copy on the target. If the file being copied is not newer, the file is not copied. The caller is not notified.

SP_COPY_WARNIFSKIP
: If the user tries to skip a file, warn them that skipping a file may affect the installation. (Used for system-critical files.)

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Remarks

SetupInstallFilesFromInfSection can be called multiple times to queue the files specified in multiple INF sections. After the queue has been successfully committed and the files have been copied, renamed, and/or deleted, **SetupInstallFromInfSection** can be called to perform registry and INI installation operations.

If a UNC directory is specified as the target directory of a file installation, you must ensure it exists before you call **SetupInstallFilesFromInfSection**. The setup functions do not check for the existence of and do not create UNC directories. If the target UNC directory does not exist, the file installation will fail.

See Also

SetupInstallServicesFromInfSection, **SetupInstallFromInfSection**

SetupInstallFromInfSection Group

Group

Group

[New - Windows NT]

The **SetupInstallFromInfSection** function carries out directives in an INF file **Install** section.

```
BOOL SetupInstallFromInfSection(
    HWND Owner,                         // optional, handle of a parent window
    HINF InfHandle,                     // handle to the INF file
    PCTSTR SectionName,                 // section of the INF file to install
    UINT Flags,                         // which lines to install from section
    HKEY RelativeKeyRoot,               // optional, key for registry installs
    PCTSTR SourceRootPath,              // optional, path for source files
    UINT CopyFlags,                     // optional, specifies copy behavior
    PSP_FILE_CALLBACK MsgHandler,       // optional, specifies callback routine
    PVOID Context,                      // optional, callback routine context
    HDEVINFO DeviceInfoSet,             // optional, device information set
    PSP_DEVINFO_DATA DeviceInfoData     // optional, device info structure
  );
```

Parameters

Owner

This optional parameter specifies the window handle of the window that owns any dialog boxes that are generated during installation, such as for disk prompting or file copying. If *Owner* is not specified, these dialog boxes become top-level windows.

InfHandle

Specifies the handle to the INF file that contains the section to be processed. The INF file must be a Windows 95- or Windows NT 4.0-style file; legacy INF files are not supported by **SetupInstallFromInfSection**.

SectionName

Supplies the name of the **Install** section in the INF file to process.

Flags

A set of flags that indicates what actions to perform. The flags can be a combination of the following:

SPINST_INIFILES

Perform INI-file operations (**UpdateInis**, **UpdateIniFields** lines in the **Install** section being processed).

SPINST_REGISTRY

Perform registry operations (**AddReg**, **DelReg** lines in the **Install** section being processed).

SPINST_INI2REG

Perform INI-file to registry operations (**Ini2Reg** lines in the **Install** section being processed).

SPINST_LOGCONFIG

This flag is only used when installing a device driver.

Perform logical configuration operations (**LogConf** lines in the **Install** section being processed). This flag is only used if *DeviceInfoSet* and *DeviceInfoData* are specified.

For more information about installing device drivers, **LogConf**, *DeviceInfoSet*, or *DeviceInfoData*, see the *DDK Programmer's Guide*.

SPINST_FILES

Perform file operations (**CopyFiles**, **DelFiles**, **RenFiles** lines in the **Install** section being processed).

SPINST_ALL
Perform all installation operations.

RelativeKeyRoot
This optional parameter must be specified if *Flags* includes SPINST_REGISTRY or SPINST_INI2REG. Specifies a handle to a registry key to be used as the root when the INF file specifies HKR as the key.

SourceRootPath
Specifies the source root for file copies. An example would be A:\ or \\*pegasus**win**install*. If *Flags* includes SPINST_FILES, and *SourceRootPath* is NULL, the system provides a default root path.

CopyFlags
This optional parameter must be specified if *Flags* includes SPINST_FILES. Specifies flags to be passed to the **SetupQueueCopySection** function when files are queued for copy. These flags may be a combination of the following values:

SP_COPY_DELETESOURCE
Delete the source file upon successful copy. The caller is not notified if the delete fails.

SP_COPY_REPLACEONLY
Copy the file only if doing so would overwrite a file at the destination path.

SP_COPY_NEWER
Examine each file being copied to see if its version resources indicate that it is not newer than an existing copy on the target.

The file version information used during version checks is that specified in the **dwFileVersionMS** and **dwFileVersionLS** members of a **VS_FIXEDFILEINFO** structure, as filled in by the Win32 version functions. If one of the files does not have version resources, or if they have identical version information, the source file is considered newer.

If the source file is not newer, and *CopyMsgHandler* is specified, the caller is notified and may cancel the copy. If *CopyMsgHandler* is not specified, the file is not copied.

SP_COPY_NOOVERWRITE
Check whether the target file exists, and, if so, notify the caller who may veto the copy. If *CopyMsgHandler* is not specified, the file is not overwritten.

SP_COPY_NODECOMP
Do not decompress the file. When this flag is set, the target file is not given the uncompressed form of the source name (if appropriate). For example, copying f:\*mips**cmd.ex_* to \\*install**temp* results in a target file of \\*install**temp**cmd.ex_*. If the SP_COPY_NODECOMP flag was not specified, the file would be decompressed and the target would be called \\*install**temp**cmd.exe*. The filename part of *DestinationName*, if specified, is stripped and replaced with the filename of the source file. When SP_COPY_NODECOMP is specified, SP_COPY_LANGUAGEAWARE and SP_COPY_NEWER are ignored.

SP_COPY_LANGUAGEAWARE
Examine each file being copied to see if its language differs from the language of any existing file already on the target. If so, and *CopyMsgHandler* is specified, the caller is notified and may cancel the copy. If *CopyMsgHandler* is not specified, the file is not copied.

SP_COPY_SOURCE_ABSOLUTE
SourceFile is a full source path. Do not look it up in the **SourceDisksNames** section of the INF file.

SP_COPY_SOURCEPATH_ABSOLUTE
SourcePathRoot is the full path part of the source file. Ignore the relative source specified in the **SourceDisksNames** section of the INF file for the source media where the file is located. This flag is ignored if SP_COPY_SOURCE_ABSOLUTE is specified.

SP_COPY_FORCE_IN_USE
If the target exists, behave as if it is in use and queue the file for copying on the next system reboot.

SP_COPY_IN_USE_NEEDS_REBOOT

If the file was in use during the copy operation inform the user that the system needs to be rebooted. This flag is only used when later calling **SetupPromptReboot** or **SetupScanFileQueue**.

SP_COPY_NO_SKIP

Do not give the user the option to skip a file.

SP_COPY_FORCE_NOOVERWRITE

Check whether the target file exists, and, if so, the file is not overwritten. The caller is not notified.

SP_COPY_FORCE_NEWER

Examine each file being copied to see if its version resources (or timestamps for non-image files) indicate that it is not newer than an existing copy on the target. If the file being copied is not newer, the file is not copied. The caller is not notified.

SP_COPY_WARNIFSKIP

If the user tries to skip a file, warn them that skipping a file may affect the installation. (Used for system-critical files.)

MsgHandler

This optional parameter must be specified if *Flags* includes SPINST-FILES. Specifies a callback function to be used when the file queue built by this routine internally is committed with the **SetupCommitFileQueue** function.

Context

This optional parameter must be specified if *Flags* includes SPINST_FIILES. Specifies a value to be passed to the callback function when the file queue built by this routine internally is committed via **SetupCommitFileQueue**.

DeviceInfoSet

This optional parameter supplies a handle to a device information set. For more information about the Device Installer setup functions, see the *DDK Programmer's Guide*.

DeviceInfoData

This optional parameter supplies a pointer to the SP_DEVINFO_DATA structure that provides a context to a specific element in the set specified by *DeviceInfoSet.* For more information about the Device Installer setup functions, see the *DDK Programmer's Guide*.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Remarks

If a UNC directory is specified as the target directory of a file copy operation, you must ensure it exists before you call **SetupInstallFromInfSection**. The setup functions do not check for the existence of and do not create UNC directories. If the target UNC directory does not exist, the file installation will fail.

See Also

SetupInstallFilesFromInfSection, **SetupInstallServicesFromInfSection**

SetupInstallServicesFromInfSection Group

Group

Group

[New - Windows NT]

The **SetupInstallServicesFromInfSection** function performs service installation and deletion operations specified in a **Service Install** section of an INF file.

```
BOOL SetupInstallServicesFromInfSection(
    HINF InfHandle,            // handle to the open INF file
    PCTSTR SectionName,        // section to install
    DWORD Flags                // controls installation procedure
  );
```

Parameters

InfHandle
Supplies the handle of an INF file that contains a **Service Install** section.

SectionName
Supplies the name of the **Service Install** section to process.

Flags
A flag that controls the installation.

SPSVCINST_TAGTOFRONT
Move the service's tag to the front of its group order list.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

See Also

SetupInstallFromInfSection, **SetupInstallFilesFromInfSection**

SetupIterateCabinet Group

Group

Group

[New - Windows NT]

The **SetupIterateCabinet** function iterates through all the files in a cabinet and sends a notification to a callback function for each file found.

```
BOOL SetupIterateCabinet(
    PCTSTR CabinetFile,                  // name of the cabinet file
    DWORD Reserved,                      // this parameter is not used
    PSP_FILE_CALLBACK MsgHandler,        // callback routine to use
    PVOID Context                        // callback routine context
  );
```

Parameters

CabinetFile

Specifies the cabinet (.CAB) file to iterate through.

Reserved

Not currently used.

MsgHandler

Supplies a pointer to a routine that will process the notifications **SetupIterateCabinet** returns as it iterates through the files in the cabinet file. The callback routine may then return a value specifying whether to decompress, copy, or skip the file.

Context

Specifies the context value that is passed into the routine specified in MsgHandler. This enables the callback routine to track values between notifications, without having to use global variables.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

SetupLogFile Group

Group

Group

[New - Windows NT]

The **SetupLogFile** function adds an entry to the log file.

```
BOOL SetupLogFile(
    HSPFILELOG FileLogHandle,     // handle to the log file
    PCTSTR LogSectionName,        // optional, name to group files by
    PCTSTR SourceFileName,        // filename on source media
    PCTSTR TargetFileName,        // filename in target directory
    DWORD Checksum,               // optional, 32-bit checksum value
    PCTSTR   DiskTagfile,         // optional, source media tag file
    PCTSTR DiskDescription,       // optional, media description
    PCTSTR OtherInfo,             // optional, additional information
    DWORD Flags                   // indicates whether OEM file
  );
```

Parameters

FileLogHandle

Supplies the handle to the file log as returned by **SetupInitializeFileLog**. The caller must not have passed SPFILELOG_QUERYONLY when the log file was initialized.

LogSectionName

This optional parameter supplies the name for a logical grouping of names within the log file. Required if SPFILELOG_SYSTEMLOG was not passed when the file log was initialized. Otherwise, *LogSectionName* is optional.

SourceFileName

Supplies the name of the file as it exists on the source media from which it was installed. This name should be in whatever format is meaningful to the caller.

TargetFileName

Supplies the name of the file as it exists on the target. This name should be in whatever format is meaningful to the caller.

Checksum

This optional parameter supplies a 32-bit checksum value. Required for the system log.

DiskTagfile

This optional parameter specifies the tagfile for the media from which the file was installed. Required for the system log if SPFILELOG_OEMFILE is specified. Ignored for the system log if SPFILELOG_OEMFILE is not specified.

DiskDescription

This optional parameter provides the human-readable description of the media from which the file was installed. Required for the system log if SPFILELOG_OEMFILE is specified in the *Flags* parameter. Ignored for the system log if SPFILELOG_OEMFILE is not specified in the *Flags* parameter.

OtherInfo

This optional parameter supplies additional information to be associated with the file.

Flags

This parameter can be SPFILELOG_OEMFILE, which is meaningful only for the system log and indicates that the file is not supplied by Microsoft. This parameter can be used to convert an existing file's entry, such as when an OEM overwrites a Microsoft-supplied system file.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

See Also

SetupRemoveFileLogEntry

SetupOpenAppendInfFile Group

Group

Group

[New - Windows NT]

The **SetupOpenAppendInfFile** function appends the information in an INF file to an INF file previously opened by **SetupOpenInfFile**.

```
BOOL SetupOpenAppendInfFile(
    PCTSTR FileName,        // optional, name of the file to append
    HINF InfHandle,         // handle of the file to append to
    PUINT ErrorLine         // optional, receives error information
  );
```

Parameters

FileName

If not NULL, *FileName* points to a null-terminated string containing the name (and optionally the path) of the INF file to be opened. If the filename does not contain path separator characters, it is searched for, first in the *%windir%\inf* directory, and then in the *%windir%\system32* directory. If the filename contains path separator characters, it is assumed to be a full path specification and no further processing is performed on it. If *FileName* is NULL, the INF filename is retrieved from the LayoutFile value of the **Version** section in the existing INF file. The same search logic is applied to the filename retrieved from the LayoutFile key.

InfHandle

Existing INF handle to which this INF file will be appended.

ErrorLine

An optional parameter that points to a caller-supplied variable to which this function returns the (1-based) line number where an error occurred during loading of the INF file. This value is generally reliable only if **GetLastError** does not return ERROR_NOT_ENOUGH_MEMORY. If an out-of-memory condition does occur, *ErrorLine* may be 0.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

If *FileName* was not specified and there was no LayoutFile value in the **Version** section of the existing INF File, **GetLastError** returns ERROR_INVALID_DATA.

Remarks

This function can only be called for Windows 95- or Windows NT 4.0-style INF files. Otherwise, the function returns FALSE and **GetLastError** will return ERROR_INVALID_PARAMETER. The main purpose of this function is to combine an INF file with the source file location information contained in the file specified in the LayoutFile entry of the **Version** section (typically, LAYOUT.INF).

See Also

SetupOpenInfFile, **SetupCloseInfFile**

SetupOpenFileQueue Group

Group

Group

[New - Windows NT]

The **SetupOpenFileQueue** function creates a setup file queue.

```
HSPFILEQ SetupOpenFileQueue(
    VOID      // takes no parameters
  );
```

Parameters

None.

Return Values

If the function succeeds, it returns a handle to a setup file queue. If there is not enough memory to create the queue, the function returns INVALID_HANDLE_VALUE.

Remarks

After the file queue has been committed and is no longer needed, **SetupCloseFileQueue** should be called to release the resources allocated during **SetupOpenFileQueue**.

See Also

SetupCloseFileQueue, **SetupCommitFileQueue**, **SetupInstallFile**, **SetupQueueCopy**, **SetupQueueDelete**, **SetupQueueRename**

SetupOpenInfFile Group

Group

Group

[New - Windows NT]

The **SetupOpenInfFile** function opens an INF file and returns a handle to it.

```
HINF SetupOpenInfFile(
    PCTSTR FileName,        // name of the INF to open
    PCTSTR InfClass,        // optional, the class of the INF file
    DWORD InfStyle,         // specifies the style of the INF file
    PUINT ErrorLine         // optional, receives error information
  );
```

Parameters

FileName

Points to a null-terminated string containing the name (and optional path) of the INF file to be opened. If the filename does not contain path separator characters, it is searched for, first in the *%windir%\inf* directory, and then in the *%windir%\system32* directory. If the filename contains path separator characters, it is assumed to be a full path specification and no further processing is performed on it.

InfClass

An optional parameter that points to a null-terminated string containing the class of INF file desired. For legacy INF files, this string must match the type specified in the OptionType value of the **Identification** section in the INF file (for example, OptionType=NetAdapter). For Windows 95- or Windows NT 4.0-style INF files, this string must match the Class value of the **Version** section (for example, Class=Net). If there is no entry in the Class value, but there is an entry for ClassGUID in the **Version** section, the corresponding class name for that GUID is retrieved and used for the comparison.

InfStyle

Specifies the style of INF file to open. May be a combination of the following flags:

INF_STYLE_OLDNT

Windows NT 3.x script-based INF files.

INF_STYLE_WIN4

Windows 95- or Windows NT 4.0-style INF files.

ErrorLine

An optional parameter that points to a caller-supplied variable to which this function returns the (1-based) line number where an error occurred during loading of the INF file. This value is generally reliable only if **GetLastError** does not return ERROR_NOT_ENOUGH_MEMORY. If an out-of-memory condition does occur, *ErrorLine* may be 0.

Return Values

The function returns a handle to the opened INF file if it is successful. Otherwise, the return value is INVALID_HANDLE_VALUE. Extended error information can be retrieved by a call to **GetLastError**.

Remarks

If the load fails because the INF file type does not match *InfClass*, the function returns FALSE and a call to **GetLastError** returns ERROR_CLASS_MISMATCH.

If multiple INF file styles are specified, the style of the INF file opened can be determined by calling the **SetupGetInfInformation** function.

Since there may be more than one class GUID with the same class name, callers interested in INF files of a particular class (that is, a particular class GUID) should retrieve the ClassGUID value from the INF file by calling **SetupQueryInfVersionInformation**.

See Also

SetupOpenAppendInfFile, **SetupCloseInfFile**, **SetupGetInfInformation**

SetupOpenMasterInf Group

Group

Group

[New - Windows NT]

The **SetupOpenMasterInf** function opens the master INF file that contains file and layout information for files shipped with Windows NT.

```
HINF SetupOpenMasterInf(
    VOID      // takes no parameters
  );
```

Parameters

None.

Return Values

If **SetupOpenMasterInf** is successful, it returns a handle to the opened INF file that contains file/layout information for files shipped with Windows NT. Otherwise, the return value is INVALID_HANDLE_VALUE. To get extended error information, call **GetLastError**.

See Also

SetupOpenInfFile, **SetupOpenAppendInfFile**

SetupPromptForDisk Group

Group

Group

[New - Windows NT]

The **SetupPromptForDisk** function displays a dialog box that prompts the user for a disk.

```
UINT SetupPromptForDisk(
    HWND hwndParent,                // parent window of the dialog box
    PCTSTR DialogTitle,             // optional, title of the dialog box
    PCTSTR DiskName,                // optional, name of disk to insert
    INPCTSTR PathToSource,          // optional, expected source path
    PCTSTR FileSought,              // name of file needed
    PCTSTR TagFile,                 // optional, source media tag file
    DWORD DiskPromptStyle,          // specifies dialog box behavior
    PTSTR PathBuffer,               // receives the source location
    DWORD PathBufferSize,           // size of the supplied buffer
    PDWORD PathRequiredSize         // optional, buffer size needed
  );
```

Parameters

hwndParent

Handle to the parent window for this dialog box.

DialogTitle

This optional parameter points to a null-terminated string specifying the dialog title. If this parameter is NULL, the default of ""%s--Files Needed"" (localized) is used. The "%s" is replaced with the text retrieved from the parent window. If no text is retrieved from the parent window, the title is "Files Needed".

DiskName

This optional parameter points to a null-terminated string specifying the name of the disk to insert. If this parameter is NULL, the default "(Unknown)" (localized) is used.

PathToSource

This optional parameter points to a null-terminated string specifying the path part of the expected location of the file, for example, F:\mips. If not specified, the path where **SetupPromptForDisk** most recently successfully located a file is used. If that list is empty, a system default is used.

FileSought

Pointer to a null-terminated string specifying the name of the file needed (filename part only). The filename is displayed if the user clicks on the **Browse** button. This routine looks for the file using its compressed form names; therefore, you can pass *cmd.exe* and not worry that the file actually exists as *cmd.ex_* on the source media.

TagFile

This optional parameter points to a null-terminated string specifying a tag file (filename part only) that identifies the presence of a particular removable media volume. If the currently selected path would place the file on removable media and a tag file is specified, **SetupPromptForDisk** looks for the tag file at the root of the drive to determine whether to continue.

For example, if *PathToSource* is A:\i386, the tagfile is *disk1.tag*, and the user types B:\i386 into the edit control of the prompt dialog box, the routine looks for B:\disk1.tag to determine whether to continue. If the tag file is not found, the function looks for the tagfile using *PathToSource*.

If a tag file is not specified, removable media works just like non-removable media and *FileSought* is looked for before continuing.

DiskPromptStyle

Specifies the behavior of the dialog box. This can be a combination of the following flags:

IDF_CHECKFIRST

Check for the file/disk before displaying the prompt dialog box, and, if present, return DPROMPT_SUCCESS immediately.

IDF_NOBEEP

Prevent the dialog box from beeping to get the user's attention when it first appears.

IDF_NOBROWSE

Do not display the browse option.

IDF_NOCOMPRESSED

Do not check for compressed versions of the source file.

IDF_NODETAILS

Do not display detail information.

IDF_NOFOREGROUND

Prevent the dialog box from becoming the foreground window.

IDF_NOSKIP

Do not display the skip option.

IDF_OEMDISK

Prompt for a disk supplied by a hardware manufacturer.

IDF_WARNIFSKIP

Warn the user that skipping a file may affect the installation.

PathBuffer

This optional parameter points to a caller-supplied buffer that, upon return, receives the path (no filename) of the location specified by the user through the dialog box.

PathBufferSize

Specifies the size of the buffer pointed to by *PathBuffer*. It should be at least MAX_PATH long.

PathRequiredSize

This optional parameter points to a caller-supplied variable that receives the required size for *PathBuffer*.

Return Values

The function returns one of the following values:

DPROMPT_SUCCESS

The requested disk/file is present and accessible. If *PathBuffer* was specified, it contains the path to the file (not including the filename).

DPROMPT_CANCEL

The user clicked on the **Cancel** button.

DPROMPT_SKIPFILE

The user clicked on the **Skip File** button.

DPROMPT_BUFFERTOOSMALL

The provided *PathBuffer* is too small. Check *PathRequiredSize* for the actual size needed for the buffer.

DPROMPT_OUTOFMEMORY

There is insufficient memory to process the request.

Remarks

For the Unicode version of this function, the buffer sizes *ReturnBufferSize* and *RequiredSize* are specified in number of characters. This number includes the null terminator. For the ANSI version of this function, the sizes are specified in number of bytes.

If this function is called with a *ReturnBuffer* of NULL and a ReturnBufferSize of zero, the function puts the buffer size needed to hold the specified data into the variable pointed to by *RequiredSize*. If the function succeeds in this, the return value is NO_ERROR. Otherwise, the return value is one of the values specified preceding.

Thus, you can call the function once to get the required buffer size, allocate the necessary memory, and then call the function a second time to retrieve the data. Using this technique, you can avoid errors due to an insufficient buffer size.

See Also

SetupCopyError, **SetupDeleteError**, **SetupRenameError**

SetupPromptReboot Group

Group

Group

[New - Windows NT]

The **SetupPromptReboot** function asks the user if he wants to reboot the system, optionally dependent on whether any files in a committed file queue were in use during a file operation. If the user answers "yes" to the prompt, shutdown is initiated before this routine returns.

```
INT SetupPromptReboot(
    HSPFILEQ FileQueue,      // optional, handle to a file queue
    HWND Owner,              // parent window of this dialog box
    BOOL ScanOnly            // optional, do not prompt user
  );
```

Parameters

FileQueue

This optional parameter supplies a handle to the file queue upon which to base the decision about whether shutdown is necessary. If *FileQueue* is not specified, **SetupPromptReboot** assumes shutdown is necessary and asks the user what to do.

Owner

Supplies the handle for the parent window to own windows created by this function.

ScanOnly

This optional parameter enables you to specify whether or not to prompt the user when **SetupPromptReboot** is called.

If TRUE, the user is never asked about rebooting, and system shutdown is not initiated. In this case, *FileQueue* must be specified. If *ScanOnly* is FALSE, the user is asked about rebooting, as previously described.

Use *ScanOnly* to determine if shutdown is necessary separately from actually initiating a shutdown.

Return Values

The function returns a combination of the following flags or -1 if an error occurred:

SPFILEQ_FILE_IN_USE

At least one file was in use during the queue commit process and there are delayed file operations pending. This flag will only be set if *FileQueue* is specified.

SPFILEQ_REBOOT_RECOMMENDED

The system should be rebooted. Depending on other flags and user response to the shutdown query, shutdown may be underway.

SPFILEQ_REBOOT_IN_PROGRESS

System shutdown is in progress.

See Also

SetupPromptForDisk

SetupQueryFileLog Group

Group

Group

[New - Windows NT]

The **SetupQueryFileLog** function returns information from a setup file log.

```
BOOL SetupQueryFileLog(
    HSPFILELOG FileLogHandle,       // handle to the log file
    PCTSTR LogSectionName,          // optional, name to group by
    PCTSTR TargetFileName,          // name of target file
    SetupFileLogInfo DesiredInfo,   // specifies info to return
    PTSTR DataOut,                  // optional, receives info
    DWORD ReturnBufferSize,         // size of supplied buffer
    PDWORD RequiredSize             // optional, needed buffer size
  );
```

Parameters

FileLogHandle

Supplies the handle to the file log as returned by **SetupInitializeLogFile**.

LogSectionName

This optional parameter supplies the section name for the log file in a format that is meaningful to the caller. Required for non-system logs. If no *LogSectionName* is specified for a system log, a default is supplied.

TargetFileName

Supplies the name of the file for which log information is desired.

DesiredInfo

Indicates what information should be returned to the *DataOut* buffer. It can take one of the following enumerated values:

Value	Meaning
SetupFileLogSourceFilename	The name of the source file as it exists on the source media
SetupFileLogChecksum	A 32-bit checksum value used by the system log
SetupFileLogDiskTagfile	The name of the tag file of the media source containing the source file
SetupFileLogDiskDescription	The human-readable description of the media where the source file resides
SetupFileLogOtherInfo	Additional information associated with the logged file

DataOut

This optional parameter points to a caller-supplied buffer in which this function returns the requested information for the file. Not all information is provided for every file. An error is not returned if an empty entry for the file exists in the log.

ReturnBufferSize

Supplies the size of the buffer, pointed to by *DataOut*. If the buffer is too small and *DataOut* is specified, data is not stored in the buffer and the function returns FALSE. If *DataOut* is not specified, the *ReturnBufferSize* parameter is ignored.

RequiredSize

This optional parameter points to a caller-supplied variable in which this function returns the required size of *DataOut*.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Remarks

For the Unicode version of this function, the buffer sizes *ReturnBufferSize*, and *ReturnRequiredSize* are specified in number of characters. This number includes the null terminator. For the ANSI version of this function, the sizes are specified in number of bytes.

If this function is called with a *ReturnBuffer* of NULL and a ReturnBufferSize of zero, the function puts the buffer size needed to hold the specified data into the variable pointed to by *RequiredSize*. If the function succeeds in this, the return value is TRUE. Otherwise, the return value is FALSE and extended error information can be obtained by calling **GetLastError**.

Thus, you can call the function once to get the required buffer size, allocate the necessary memory, and then call the function a second time to retrieve the data. Using this technique, you can avoid errors due to an insufficient buffer size.

Remarks

If the value of *DesiredInfo* is greater than *SetupFileLogOtherInfo* the function will fail, and **GetLastError** will return ERROR_INVALID_PARAMETER.

See Also

SetupLogFile

SetupQueryInfFileInformation Group

Group

Group

[New - Windows NT]

The **SetupQueryInfFileInformation** function returns an INF filename from an **SP_INF_INFORMATION** structure to a caller-supplied buffer.

```
BOOL SetupQueryInfFileInformation(
    PSP_INF_INFORMATION InfInformation,    // structure that contains the INF info
    UINT InfIndex,                         // index of the file to investigate
    PTSTR ReturnBuffer,                    // optional, receives the information
    DWORD ReturnBufferSize,                // size of the supplied buffer
    PDWORD Required Size                   // optional, buffer size needed
  );
```

Parameters

InfInformation
: Points to an **SP_INF_INFORMATION** structure returned from a call to the **SetupGetInfInformation** function.

InfIndex
: The index of the constituent INF filename to retrieve. This index can be in the range [0, *InfInformation.InfCount*). Meaning that the values zero through, but not including, *InfInformation.InfCount* are valid.

ReturnBuffer
: If not NULL, *ReturnBuffer* supplies a pointer to a caller-supplied buffer in which this function returns the full INF filename.

ReturnBufferSize
: Specifies the size of the buffer pointed to by the *ReturnBuffer* parameter.

RequiredSize
: If not NULL, points to a caller-supplied variable in which this function returns the required size for the buffer pointed to by the *ReturnBuffer* parameter. If *ReturnBuffer* is specified and the actual size is larger than *ReturnBufferSize*, the function fails and a call to **GetLastError** returns ERROR_INSUFFICIENT_BUFFER.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Remarks

For the Unicode version of this function, the buffer sizes *PathBufferSize*, and *PathRequiredSize* are specified in number of characters. This number includes the null terminator. For the ANSI version of this function, the sizes are specified in number of bytes.

If this function is called with a *ReturnBuffer* of NULL and a ReturnBufferSize of zero, the function puts the buffer size needed to hold the specified data into the variable pointed to by *RequiredSize*. If the function succeeds in this, the return value is TRUE. Otherwise, the return value is FALSE and extended error information can be obtained by calling **GetLastError**.

Thus, you can call the function once to get the required buffer size, allocate the necessary memory, and then call the function a second time to retrieve the data. Using this technique, you can avoid errors due to an insufficient buffer size.

See Also

SetupGetInfInformation, **SetupQueryInfVersionInformation**

SetupQueryInfVersionInformation Group

Group

Group

[New - Windows NT]

The **SetupQueryInfVersionInformation** function returns INF file version information from an **SP_INF_INFORMATION** structure to a caller-supplied buffer.

```
BOOL SetupQueryInfVersionInformation(
    PSP_INF_INFORMATION InfInformation,      // structure that contains INF info
    UINT InfIndex,                           // index of the file to investigate
    PCTSTR Key,                              // optional, the key to retrieve
    PTSTR ReturnBuffer,                      // optional, receives the version info
    DWORD ReturnBufferSize,                  // size of the supplied buffer
    PDWORD RequiredSize                      // optional, buffer size needed
  );
```

Parameters

InfInformation

Points to an **SP_INF_INFORMATION** structure previously returned from a call to the **SetupGetInfInformation** function.

InfIndex

Specifies the index of the constituent INF file to retrieve version information from. This index can be in the range [0, *InfInformation.InfCount*).   Meaning that the values zero through, but not including, *InfInformation.InfCount* are valid.

Key

An optional parameter that points to a null-terminated string containing the key name whose associated string is to be retrieved. If this parameter is NULL, all resource keys are copied to the supplied buffer. Each string is null-terminated, with an extra null at the end of the list.

ReturnBuffer

If not NULL, *ReturnBuffer* points to a call-supplied character buffer in which this function returns the INF file style.

ReturnBufferSize

Specifies the size of the buffer pointed to by the *ReturnBuffer* parameter.

RequiredSize

If not NULL, points to a caller-supplied variable in which this function returns the size required for the buffer pointed to by the *ReturnBuffer* parameter. If *ReturnBuffer* is specified and the actual size is larger than the value specified by *ReturnBufferSize*, the function fails and a call to **GetLastError** returns ERROR_INSUFFICIENT_BUFFER.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Remarks

For the Unicode version of this function, the buffer sizes *ReturnBufferSize*, and *ReturnRequiredSize* are specified in number of characters. This number includes the null terminator. For the ANSI version of this function, the sizes are specified in number of bytes.

If this function is called with a *ReturnBuffer* of NULL and a ReturnBufferSize of zero, the function puts the buffer size needed to hold the specified data into the variable pointed to by *RequiredSize*. If the function succeeds in this, the return value is TRUE. Otherwise, the return value is FALSE and extended error information can be obtained by calling **GetLastError**.

Thus, you can call the function once to get the required buffer size, allocate the necessary memory, and then call the function a second time to retrieve the data. Using this technique, you can avoid errors due to an insufficient buffer size.

If **SetupQueryInfVersionInformation** is called on a Windows NT 3.x INF file, then version information is generated from the Windows NT 3.x INF file in the following manner:

1. The OptionType key in the **Identification** section of the Windows NT 3.*x* INF file is returned as the Class key value.
2. The FileType key in the **Signature** section of the Windows NT 3.*x* INF file becomes the Signature key value.
3. If the value of the FileType key of the Windows NT 3.*x* INF file is MICROSOFT_FILE, then the Provider key value is set to "Microsoft".

The following table summarizes how the information is translated before it is passed into the **SP_INF_INFORMATION** structure.

Windows NT 3.x information	**mapped to Windows 95 or Windows NT 4.0 data**
[Identification] OptionType = Mouse	[Version] Class=Mouse
[Signature] FileType = MICROSOFT_FILE	Signature=MICROSOFT_FILE
(if the FileType is MICROSOFT_FILE)	Provider="Microsoft"

See Also

SetupGetInfInformation, **SetupQueryInfFileInformation**

SetupQuerySourceList Group

Group

Group

[New - Windows NT]

The **SetupQuerySourceList** function queries the current list of installation sources. The list is built from the system and user-specific lists, and potentially overridden by a temporary list (see SetupSetSourceList).

```
BOOL SetupQuerySourceList(
    DWORD Flags,       // specifies the list to query
    PCTSTR **List,     // receives an array of sources
    PUINT Count        // number of sources in the array
  );
```

Parameters

Flags

These flags specify which list to query. This parameter can be any combination of the following values:

SRCLIST_SYSTEM

Query the system list.

SRCLIST_USER

Query the per-user list.

Note If the system and the user lists are both retrieved, they are merged with those items in the system list that appear first.

If none of the preceding flags are specified, the entire current (merged) list is returned.

SRCLIST_NOSTRIPPLATFORM

Normally, all paths are stripped of a platform-specific component if it is the final component. For example, a path stored in the registry as f:\mips is returned as f:\. If this flag is specified, the platform-specific component is not stripped.

List

Supplies a pointer to a caller-supplied variable in which this function returns a pointer to an array of sources. The caller must free this array with a call to SetupFreeSourceList.

Count

Supplies a pointer to a caller-supplied variable in which this function returns the number of sources in the list.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call GetLastError.

See Also

SetupSetSourceList

SetupQueueCopy Group

Group

Group

[New - Windows NT]

The **SetupQueueCopy** function adds a single file copy operation to a setup file queue.

```
BOOL SetupQueueCopy(
  HSPFILEQ QueueHandle,          // handle to the file queue
  PCTSTR SourceRootPath,         // path to the source file
  PCTSTR SourcePath,             // optional, additional path info
  PCTSTR SourceFileName,         // name of file to copy
  PCTSTR SourceDescription,      // optional, source description
  PCTSTR SourceTagFile,          // optional, source media tag file
  PCTSTR TargetDirectory,        // directory to copy file to
  PCTSTR TargetFileName,         // optional, name for copied file
  DWORD CopyStyle                // specifies copy behavior
 );
```

Parameters

QueueHandle
Supplies a handle to a setup file queue, as returned by **SetupOpenFileQueue**.

SourceRootPath
Supplies the root of the source for this copy, such as A:\.

SourcePath
This optional parameter supplies the path relative to *SourceRootPath* where the file can be found.

SourceFileName
Supplies the filename part of the file to be copied.

SourceDescription
This optional parameter supplies a description of the source media to be used during disk prompts.

SourceTagFile
This optional parameter supplies a tag file whose presence at *SourceRootPath* indicates the presence of the source media. If not specified, the file itself will be used as the tag file if required.

TargetDirectory
Supplies the directory where the file is to be copied.

TargetFileName
This optional parameter supplies the name of the target file. If not specified, the target file will have the same name as the source file.

CopyStyle
Flags that control the behavior of the file copy operation. These flags may be a combination of the following values:

SP_COPY_DELETESOURCE
Delete the source file upon successful copy. The caller is not notified if the delete fails.

SP_COPY_REPLACEONLY
Copy the file only if doing so would overwrite a file at the destination path. The caller is not notified.

SP_COPY_NEWER
Examine each file being copied to see if its version resources indicate that it is not newer than an existing copy on the target.

The file version information used during version checks is that specified in the **dwFileVersionMS** and **dwFileVersionLS** members of a **VS_FIXEDFILEINFO** structure, as filled in by the Win32 version functions. If one of the files does not have version resources, or if they have identical version information, the source file is considered newer.

If the source file is not newer, and *CopyMsgHandler* is specified, the caller is notified and may cancel the copy. If *CopyMsgHandler* is not specified, the file is not copied.

SP_COPY_NOOVERWRITE
Check whether the target file exists, and, if so, notify the caller who may veto the copy. If *CopyMsgHandler* is not specified, the file is not overwritten.

SP_COPY_NODECOMP
Do not decompress the file. When this flag is set, the target file is not given the uncompressed form of the source name (if appropriate). For example, copying f:*\mips\cmd.ex_* to *\\install\temp* results in a target file of *\\install\temp\cmd.ex_*. If the SP_COPY_NODECOMP flag was not specified, the file would be decompressed and the target would be called *\\install\temp\cmd.exe*. The filename part of *DestinationName*, if specified, is stripped and replaced with the filename of the source file. When SP_COPY_NODECOMP is specified, SP_COPY_LANGUAGEAWARE and SP_COPY_NEWER are ignored.

SP_COPY_LANGUAGEAWARE
Examine each file being copied to see if its language differs from the language of any existing file already on the target. If so, and *CopyMsgHandler* is specified, the caller is notified and may cancel the copy. If *CopyMsgHandler* is not specified, the file is not copied.

SP_COPY_SOURCE_ABSOLUTE
SourceFile is a full source path. Do not look it up in the **SourceDisksNames** section of the INF file.

SP_COPY_SOURCEPATH_ABSOLUTE
SourcePathRoot is the full path part of the source file. Ignore the relative source specified in the **SourceDisksNames** section of the INF file for the source media where the file is located. This flag is ignored if SP_COPY_SOURCE_ABSOLUTE is specified.

SP_COPY_FORCE_IN_USE
If the target exists, behave as if it is in use and queue the file for copying on the next system reboot.

SP_COPY_IN_USE_NEEDS_REBOOT
If the file was in use during the copy operation, alert the user that the system needs to be rebooted.

SP_COPY_NO_SKIP
Do not give the user the option to skip a file.

SP_COPY_FORCE_NOOVERWRITE
Check whether the target file exists, and, if so, the file is not overwritten. The caller is not notified.

SP_COPY_FORCE_NEWER
Examine each file being copied to see if its version resources (or timestamps for non-image files) indicate that it is not newer than an existing copy on the target. If the file being copied is not newer, the file is not copied. The caller is not notified.

SP_COPY_WARNIFSKIP
If the user tries to skip a file, warn them that skipping a file may affect the installation. (Used for system-critical files.)

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Remarks

If a UNC directory is specified as the target directory of a file copy operation, you must ensure it exists before the queue is committed. The setup functions do not check for the existence of and do not create UNC directories. If the target UNC directory does not exist, the file copy will fail.

See Also

SetupQueueCopySection, SetupQueueDefaultCopy, SetupQueueDelete, SetupQueueRename

SetupQueueCopySection Group

Group

Group

[New - Windows NT]

The **SetupQueueCopySection** function places all the files in a section of an INF file in a setup queue for copying. The section must be in the correct **Copy Files** format and the INF file must contain **SourceDisksFiles** and **SourceDisksNames** sections (or have had the INF files containing those sections appended).

```
BOOL SetupQueueCopySection(
    HSPFILEQ QueueHandle,      // handle of the file queue
    PCTSTR SourceRootPath,     // path to the source media
    HINF InfHandle,            // handle to the master INF file
    HINF ListInfHandle,        // optional, handle to section INF
    PCTSTR Section,            // INF section w/files to copy
    DWORD CopyStyle            // specifies copy behavior
  );
```

Parameters

QueueHandle
: Supplies a handle to a setup file queue, as returned by **SetupOpenFileQueue**.

SourceRootPath
: Supplies the root of the source for this copy, such as A:\.

InfHandle
: This optional parameter points to the handle of an open INF file that contains the **SourceDisksFiles** and **SourceDisksNames** sections, and, if *ListInfHandle* is not specified, contains the section names. If platform-specific sections exist for the user's system (for example, **SourceDisksNames.mips** and **SourceDisksFiles.mips**), the platform-specific section will be used. This handle must be for a Windows 95- or Windows NT 4.0-style INF file.

ListInfHandle
: If specified, supplies a handle to an open INF file that contains the section to queue for copying. If *ListInfHandle* is not specified, *InfHandle* is assumed to contain the section.

Section
: Supplies the name of the section to be queued for copy.

CopyStyle
: Flags that control the behavior of the file copy operation. These flags may be a combination of the following values:

SP_COPY_DELETESOURCE
: Delete the source file upon successful copy. The caller is not notified if the delete fails.

SP_COPY_REPLACEONLY
: Copy the file only if doing so would overwrite a file at the destination path.

SP_COPY_NEWER
: Examine each file being copied to see if its version resources indicate that it is not newer than an existing copy on the target.

The file version information used during version checks is that specified in the **dwFileVersionMS** and **dwFileVersionLS** members of a **VS_FIXEDFILEINFO** structure, as filled in by the Win32 version functions. If one of the files does not have version resources, or if they have identical version information, the source file is considered newer.

If the source file is not newer, and *CopyMsgHandler* is specified, the caller is notified and may cancel the copy. If *CopyMsgHandler* is not specified, the file is not copied.

SP_COPY_NOOVERWRITE
: Check whether the target file exists, and, if so, notify the caller who may veto the copy. If *CopyMsgHandler* is not specified, the file is not overwritten.

SP_COPY_NODECOMP
: Do not decompress the file. When this flag is set, the target file is not given the uncompressed form of the source name (if appropriate). For example, copying f:*\mips\cmd.ex_* to *\\install\temp* results in a target file of *\\install\temp\cmd.ex_*. If the SP_COPY_NODECOMP flag was not specified, the file would be decompressed and the target would be called *\\install\temp\cmd.exe*. The filename part of *DestinationName*, if specified, is stripped and replaced with the filename of the source file. When SP_COPY_NODECOMP is specified, SP_COPY_LANGUAGEAWARE and SP_COPY_NEWER are ignored.

SP_COPY_LANGUAGEAWARE
: Examine each file being copied to see if its language differs from the language of any existing file already on the target. If so, and *CopyMsgHandler* is specified, the caller is notified and may cancel the copy. If *CopyMsgHandler* is not specified, the file is not copied.

SP_COPY_SOURCE_ABSOLUTE
: *SourceFile* is a full source path. Do not look it up in the **SourceDisksNames** section of the INF file.

SP_COPY_SOURCEPATH_ABSOLUTE
: *SourcePathRoot* is the full path part of the source file. Ignore the relative source specified in the **SourceDisksNames** section of the INF file for the source media where the file is located. This flag is ignored if SP_COPY_SOURCE_ABSOLUTE is specified.

SP_COPY_FORCE_IN_USE
: If the target exists, behave as if it is in use and queue the file for copying on the next system reboot.

SP_COPY_IN_USE_NEEDS_REBOOT
: If the file was in use during the copy operation, alert the user that the system needs to be rebooted.

SP_COPY_NO_SKIP
: Do not give the user the option to skip a file.

SP_COPY_FORCE_NOOVERWRITE
: Check whether the target file exists, and, if so, the file is not overwritten. The caller is not notified.

SP_COPY_FORCE_NEWER
: Examine each file being copied to see if its version resources (or timestamps for non-image files) indicate that it is not newer than an existing copy on the target. If the file being copied is not newer, the file is not copied. The caller is not notified.

SP_COPY_WARNIFSKIP
: If the user tries to skip a file, warn them that skipping a file may affect the installation. (Used for system-critical files.)

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Remarks

If a UNC directory is specified as the target directory of a file copy operation, you must ensure it exists before the queue is committed. The setup functions do not check for the existence of and do not create UNC directories. If the target UNC directory does not exist, the file copy will fail.

See Also

SetupQueueCopy, **SetupQueueDefaultCopy**

SetupQueueDefaultCopy Group

Group

Group

[New - Windows NT]

The **SetupQueueDefaultCopy** function adds a single file to a setup file queue for copying, using the default source media and destination as specified in an INF file.

```
BOOL SetupQueueDefaultCopy(
    HSPFILEQQueueHandle,        // handle to the file queue
    HINF InfHandle,             // handle to the INF file
    PCTSTR SourceRootPath,      // path to the source media
    PCTSTR SourceFileName,      // name of the file to copy
    PCTSTR TargetFileName,      // name of the copied file
    DWORD CopyStyle             // specifies copy behavior
  );
```

Parameters

QueueHandle

Supplies a handle to a setup file queue, as returned by **SetupOpenFileQueue**.

InfHandle

Supplies a handle to an open INF file that contains the **SourceDisksFiles** and **SourceDisksNames** sections. If platform-specific sections exist for the user's system (for example, **SourceDisksNames.mips** and **SourceDisksFiles.mips**), the platform-specific section will be used. This handle must be for a Windows 95- or Windows NT 4.0-style INF file.

SourceRootPath

Supplies the root directory of the source for this copy such as A:\.

SourceFileName

Supplies the filename of the file to be copied.

TargetFileName

Supplies the filename of the target file.

CopyStyle

Flags that control the behavior of the file copy operation. These flags may be a combination of the following values:

SP_COPY_DELETESOURCE

Delete the source file upon successful copy. The caller is not notified if the delete fails.

SP_COPY_REPLACEONLY

Copy the file only if doing so would overwrite a file at the destination path.

SP_COPY_NEWER

Examine each file being copied to see if its version resources indicate that it is not newer than an existing copy on the target.

The file version information used during version checks is that specified in the **dwFileVersionMS** and **dwFileVersionLS** members of a **VS_FIXEDFILEINFO** structure, as filled in by the Win32 version functions. If one of the files does not have version resources, or if they have identical version information, the source file is considered newer.

If the source file is not newer, and *CopyMsgHandler* is specified, the caller is notified and may cancel the copy. If *CopyMsgHandler* is not specified, the file is not copied.

SP_COPY_NOOVERWRITE

Check whether the target file exists, and, if so, notify the caller who may veto the copy. If *CopyMsgHandler* is not specified, the file is not overwritten.

SP_COPY_NODECOMP
Do not decompress the file. When this flag is set, the target file is not given the uncompressed form of the source name (if appropriate). For example, copying f:*\mips\cmd.ex_* to *\\install\temp* results in a target file of *\\install\temp\cmd.ex_*. If the SP_COPY_NODECOMP flag was not specified, the file would be decompressed and the target would be called *\\install\temp\cmd.exe*. The filename part of *DestinationName*, if specified, is stripped and replaced with the filename of the source file. When SP_COPY_NODECOMP is specified, SP_COPY_LANGUAGEAWARE and SP_COPY_NEWER are ignored.

SP_COPY_LANGUAGEAWARE
Examine each file being copied to see if its language differs from the language of any existing file already on the target. If so, and *CopyMsgHandler* is specified, the caller is notified and may cancel the copy. If *CopyMsgHandler* is not specified, the file is not copied.

SP_COPY_SOURCE_ABSOLUTE
SourceFile is a full source path. Do not look it up in the **SourceDisksNames** section of the INF file.

SP_COPY_SOURCEPATH_ABSOLUTE
SourcePathRoot is the full path part of the source file. Ignore the relative source specified in the **SourceDisksNames** section of the INF file for the source media where the file is located. This flag is ignored if SP_COPY_SOURCE_ABSOLUTE is specified.

SP_COPY_FORCE_IN_USE
If the target exists, behave as if it is in use and queue the file for copying on the next system reboot.

SP_COPY_IN_USE_NEEDS_REBOOT
If the file was in use during the copy operation, alert the user that the system needs to be rebooted.

SP_COPY_NO_SKIP
Do not give the user the option to skip a file.

SP_COPY_FORCE_NOOVERWRITE
Check whether the target file exists, and, if so, the file is not overwritten. The caller is not notified.

SP_COPY_FORCE_NEWER
Examine each file being copied to see if its version resources (or timestamps for non-image files) indicate that it is not newer than an existing copy on the target. If the file being copied is not newer, the file is not copied. The caller is not notified.

SP_COPY_WARNIFSKIP
If the user tries to skip a file, warn them that skipping a file may affect the installation. (Used for system-critical files.)

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Remarks

If a UNC directory is specified as the target directory of a file copy operation, you must ensure it exists before the queue is committed. The setup functions do not check for the existence of and do not create UNC directories. If the target UNC directory does not exist, the file copy will fail.

The default destination used by this function is specified by the **DefaultDestDir** key in the **DestinationDirs** section of an INF file.

See Also

SetupQueueCopy, **SetupQueueCopySection**

SetupQueueDelete Group

Group

Group

[New - Windows NT]

The **SetupQueueDelete** function places an individual file delete operation on a setup file queue.

```
BOOL SetupQueueDelete(
  HSPFILEQ QueueHandle,      // handle to the file queue
  PCTSTR PathPart1,          // path to the file to delete
  PCTSTR PathPart2           // optional, additional path info
 );
```

Parameters

QueueHandle
Supplies a handle to a setup file queue, as returned by **SetupOpenFileQueue**.

PathPart1
Supplies the first part of the path of the file to be deleted. If *PathPart2* is not specified, *PathPart1* is the full path of the file to be deleted.

PathPart2
This optional parameter supplies the second part of the path of the file to be deleted. This is appended to *PathPart1* to form the full path of the file to be deleted. The function checks for and collapses duplicated path separators when it combines *PathPart1* and *PathPart2*.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Remarks

Since delete operations are assumed to take place on fixed media, the user will not be prompted when the queue is committed.

See Also

SetupQueueCopy, **SetupQueueDeleteSection**, **SetupQueueRename**

SetupQueueDeleteSection Group

Group

Group

[New - Windows NT]

The **SetupQueueDeleteSection** function queues all the files in a section of an INF file for deletion. The section must be in the correct **Delete Files** format and the INF file must contain a **DestinationDirs** section.

```
BOOL SetupQueueDeleteSection(
    HSPFILEQ QueueHandle,       // handle to the file queue
    HINF InfHandle,             // handle to the INF file
    HINF ListInfHandle,         // optional, handle to section INF
    PCTSTR Section              // INF section that lists the files to delete
  );
```

Parameters

QueueHandle

Supplies a handle to a setup file queue, as returned by **SetupOpenFileQueue**.

InfHandle

Supplies a handle to an open INF file that contains the **DestinationDirs** section. If *ListInfHandle* is not specified, *InfHandle* contains the section name. This handle must be for a Windows 95- or Windows NT 4.0-style INF file.

ListInfHandle

This optional parameter points to the handle of an open INF file that contains the section to queue for deletion. If *ListInfHandle* is not specified, *InfHandle* is assumed to contain the section name.

Section

Supplies the name of the section to be queued for deletion.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

See Also

SetupQueueCopySection, **SetupQueueDelete**, **SetupQueueRenameSection**

SetupQueueRename Group

Group

Group

[New - Windows NT]

The **SetupQueueRename** function places an individual file rename operation on a setup file queue.

```
BOOL SetupQueueRename(
    HSPFILEQ QueueHandle,         // handle to the file queue
    PCTSTR SourcePath,            // path to the file to rename
    PCTSTR SourceFileName,        // optional, source filename
    PCTSTR TargetPath,            // optional, new path for file
    PCTSTR TargetFileName         // optional, new name for file
  );
```

Parameters

QueueHandle

Supplies a handle to a setup file queue, as returned by **SetupOpenFileQueue**.

SourcePath

Supplies the source path of the file to be renamed. If *SourceFileName* is not specified, *SourcePath* is assumed to be the full path.

SourceFileName

This optional parameter supplies the filename part of the file to be renamed. If not specified, *SourcePath* is the full path.

TargetPath

This optional parameter supplies the target directory and the rename operation is actually a mover operation. If TargetPath is not specified, the file is renamed but remains in its current location.

TargetFileName

This optional parameter supplies the name of the new name for the source file.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Remarks

Since rename operations are assumed to take place on fixed media, the user will not be prompted when the queue is committed.

See Also

SetupQueueDelete, **SetupQueueCopy**, **SetupQueueRenameSection**

SetupQueueRenameSection Group

Group

Group

[New - Windows NT]

The **SetupQueueRenameSection** function queues a section in an INF file for renaming. The section must be in the correct rename list section format and the INF file must contain a **DestinationDirs** section.

```
BOOL SetupQueueRenameSection(
    HSPFILEQ QueueHandle,       // handle to the file queue
    HINF InfHandle,             // handle to the INF file
    HINF ListInfHandle,         // optional, section INF handle
    PCTSTR Section              // section that lists the files to rename
  );
```

Parameters

QueueHandle

Supplies a handle to a setup file queue, as returned by **SetupOpenFileQueue**.

InfHandle

A handle to the INF file that contains the **DestinationDirs** section. If *ListInfHandle* is not specified, *InfHandle* contains the section name. This handle must be for a Windows 95- or Windows NT 4.0-style INF file.

ListInfHandle

This optional parameter supplies a handle to an INF file that contains the section to queue for renaming. If *ListInfHandle* is not specified, *InfHandle* is assumed to contain the section name.

Section

Supplies the name of the section to be queued for renaming.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Remarks

You cannot queue file moves with **SetupQueueRenameSection** because the form of a rename list section limits section renaming to within the same directory.

See Also

SetupQueueCopySection, **SetupQueueDeleteSection**, **SetupQueueRename**

SetupRemoveFileLogEntry Group

Group

Group

[New - Windows NT]

The **SetupRemoveFileLogEntry** function removes an entry or section from a file log.

```
BOOL SetupRemoveFileLogEntry(
    HSPFILELOG FileLogHandle,       // handle to the log file
    PCTSTR LogSectionName,          // optional, name to group by
    PCTSTR TargetFileName           // optional, name in target dir
  );
```

Parameters

FileLogHandle
: Supplies the handle to the file log as returned by **SetupInitializeFileLog**. The caller must not have passed SPFILELOG_QUERYONLY when the log file was initialized.

LogSectionName
: This optional parameter supplies the name for a logical grouping of names within the log file. Required for non-system logs. Otherwise, *LogSectionName* is optional.

TargetFileName
: This optional parameter supplies the name of the file as it exists on the target. This name should be in whatever format is meaningful to the caller. If not specified, the section specified by LogSectionName is removed. The main section for Windows NT files cannot be removed.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

See Also

SetupLogFile

SetupRemoveFromSourceList Group

Group

Group

[New - Windows NT]

The **SetupRemoveFromSourceList** function removes a value from the list of installation sources for either the current user or the system. The system and user lists are merged at run time.

```
BOOL SetupRemoveFromSourceList(
    DWORD Flags,        // the list to remove the source from
    PCTSTR Source       // the source to remove
  );
```

Parameters

Flags

These flags specify which list to remove the source from. This parameter can be any combination of the following values:

SRCLIST_SYSTEM

Remove the source to the per-system list. The caller must be an administrator.

SRCLIST_USER

Remove the source to the per-user list.

SRCLIST_SYSIFADMIN

If the caller is an administrator, the source is removed from the per-system list; if the caller is not an administrator, the source is removed from the per-user list for the current user.

Note If a temporary list is currently in use (see **SetupSetSourceList**), the preceding flags are ignored and the source is removed from the temporary list.

SRCLIST_SUBDIRS

Remove all subdirectories of the source.

Source

Pointer to the source to remove from the list.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

See Also

SetupAddToSourceList, **SetupSetSourceList**

SetupRenameError Group

Group

Group

[New - Windows NT]

The **RenameError** function generates a dialog box that informs the user of a file renaming error.

```
UINT SetupRenameError(
    HWND hwndParent,           // parent window for this dialog box
    PCTSTR DialogTitle,        // optional, title for this dialog box
    PCTSTR SourceFile,         // source file of the rename error
    PCTSTR TargetFile,         // target file of the rename error
    UINT Win32ErrorCode,       // the error encountered
    DWORD Style                // specifies formatting for this dialog box
 );
```

Parameters

hwndParent
Handle to the parent window for this dialog box.

DialogTitle
This optional parameter points to a null-terminated string specifying the error dialog box title. If this parameter is NULL, the default title of "Rename Error" (localized) is used.

SourceFile
Pointer to a null-terminated string specifying the full path of the source file on which the operation failed.

TargetFile
Pointer to a null-terminated string specifying the full path of the target file on which the operation failed.

Win32ErrorCode
The Win32 error code encountered during the file operation. For information about Win32 error codes, see the WINERROR.H file included with the Win32 SDK.

Style
Specifies flags that control display formatting and behavior of the dialog box. This parameter can be one of the following flags:

IDF_NOBEEP
Prevent the dialog box from beeping to get the user's attention when it first appears.

IDF_NOFOREGROUND
Prevent the dialog box from becoming the foreground window.

Return Values

This function returns one of the following values:

DPROMPT_SUCCESS
The user retried the operation and it was successful.

DPROMPT_CANCEL
The user clicked on the **Cancel** button.

DPROMPT_SKIPFILE
The user clicked on the **Skip File** button.

DPROMPT_OUTOFMEMORY
There is insufficient memory to process the request.

See Also

SetupCopyError, **SetupDeleteError**, **SetupPromptForDisk**

SetupScanFileQueue Group

Group

Group

[New - Windows NT]

The **SetupScanFileQueue** function scans a setup file queue, performing an operation on each node in its copy list. The operation is specified by a set of flags. This function can be called either before or after the queue has been committed

```
BOOL SetupScanFileQueue(
  HSPFILEQ FileQueue,                     // handle to the file queue
  DWORD Flags,                            // control scan operation
  HWND Window,                            // optional, specifies a parent window
  PSP_FILE_CALLBACK CallbackRoutine,      // optional, callback routine to use
  PVOID CallbackContext,                  // optional, callback routine context
  PDWORD Result                           // receives scan result
 );
```

Parameters

FileQueue

Supplies a handle to the setup file queue whose copy list is to be scanned/iterated.

Flags

Values that control how the scan operation is carried out. Flags can be a combination of the following values:

SPQ_SCAN_FILE_PRESENCE

Determine whether all target files in the copy queue are already present on the target.

SPQ_SCAN_USE_CALLBACK

For each node of the queue, call the callback routine. If the callback routine returns non-0, queue processing stops and **SetupScanFileQueue** returns FALSE immediately.

Either SPQ_SCAN_FILE_PRESENCE, or SPQ_SCAN_USE_CALLBACK must be specified.

SPQ_INFORM_USER

If this flag is specified and all the files in the queue pass the presence/validity check, **SetupScanFileQueue** informs the user that the operation being attempted requires files but they are already present on the target. This flag is ignored if SPQ_SCAN_FILE_PRESENCE is not specified.

Window

This optional parameter specifies the window to own dialog boxes that are presented. The Window parameter is not used if the *Flags* parameter does not contain SPQ_SCAN_FILE_PRESENCE or if *Flags* does not contain SPQ_SCAN_INFORM_USER.

CallbackRoutine

This optional parameter specifies a callback function to be called on each node of the copy queue. The notification code passed to the callback function is SPFILENOTIFY_QUEUESCAN. This parameter is required if *Flags* includes SPQ_SCAN_USE_CALLBACK.

Note You must supply the callback routine specified by *CallbackRoutine*. The default queue callback routine does not support **SetupScanFileQueue**.

CallbackContext

This optional parameter points to a context that contains caller-defined data passed to the callback routine pointed to by *CallbackRoutine*.

Result

Supplies a pointer to a caller-supplied variable in which this function returns the result of the scan operation.

Return Values

The function returns TRUE if all nodes in the queue were processed.

If the SPQ_SCAN_USE_CALLBACK flag was set, the value in *Result* is 0. The callback routine specified by *CallbackRoutine* is sent the notification SPFILENOTIFY_QUEUESCAN. *CallbackRoutine.Param1* specifies a pointer to an array that contains the target path information. The pointer has been cast to an unsigned integer and must be recast to an TCHAR array of MAX_PATH elements before a callback routine can access the information. *CallbackRoutine.Param2* is set to SPQ_DELAYED_COPY if the current queue node is in use and cannot be copied until the system is rebooted. Otherwise, *CallbackRoutine.Param2* takes the value 0.

If SPQ_SCAN_USE_CALLBACK was not set, *Result* indicates whether the queue passed the presence/validity check as shown in the following table.

Result	Meaning
0	The queue failed the check or it passed the check but SPQ_SCAN_INFORM_USER was specified and the user wants new copies of the files.
1	The queue passed the check and, if SPQ_SCAN_INFORM_USER was specified, the user indicated that copying is not required. The copy queue is empty and there are no elements on the delete or rename queues, so the caller can skip queue commit.
2	The queue passed the check and, if SPQ_SCAN_INFORM_USER was specified, the user indicated that copying is not required. The copy queue is empty but there are elements on the delete or rename queues, so the call cannot skip queue commit.

The function returns FALSE if an error occurred or the callback function returned non-0. If *Result* is non-0, it is the value returned by the callback function that stopped queue processing. If *Result* is 0, extended error information can be retrieved by a call to **GetLastError**.

See Also

SetupCommitFileQueue, **SetupDefaultQueueCallback**

SetupSetDirectoryId Group

Group

Group

[New - Windows NT]

The **SetupSetDirectoryId** function associates a directory identifier in an INF file with a particular directory.

```
BOOL SetupSetDirectoryId(
    HINF InfHandle,           // handle to the INF file
    DWORD Id,                 // optional, DIRID to assign to Directory
    PCTSTR Directory          // optional, directory to map to identifier
  );
```

Parameters

InfHandle

Handle for a loaded INF file.

Id

This optional parameter supplies the directory identifier (DIRID) to use for the association. This DIRID must be greater than or equal to DIRID_USER. If an association already exists for this DIRID, it is overwritten. If *Id* is not specified, the *Directory* parameter is ignored and the current set of user-defined DIRIDs is deleted.

Directory

This optional parameter supplies the directory path to associate with *Id*. If *Directory* is NULL, any directory associated with *Id* is unassociated. No error results if *Id* is not currently associated with a directory.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

ERROR_NOT_ENOUGH_MEMORY

Indicates that a memory allocation failed.

ERROR_INVALID_PARAMETER

The *Id* parameter is not greater than or equal to DIRID_USER or *Directory* is not a valid string.

Remarks

SetupSetDirectoryId can be used prior to queueing file copy operations, to specify a target location that is only known at runtime.

After setting the directory identifier, this function traverses all appended INF files, and, if any of them have unresolved string substitutions, attempts to re-apply string substitution to them based on the new DIRID mapping. Because of this, some INF values may change after calling **SetupSetDirectoryId**.

DIRID_ABSOLUTE_16BIT is not a valid value for *Id*. This ensures compatibility with 16-bit setupx.

SetupSetPlatformPathOverride Group

Group

Group

[New - Windows NT]

The **SetupSetPlatformPathOverride** function sets the platform path override or removes it if none is specified.

```
BOOL SetupSetPlatformPathOverride(
    PCTSTR Override       // optional, platform replacement string
  );
```

Parameters

Override

This optional parameter points to a string that contains the replacement platform information. For example, "mips", "alpha", "ppc", or "i386".

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

If **GetLastError** returns ERROR_NOT_ENOUGH_MEMORY, **SetupSetPlatformPathOverride** was unable to store the *Override* string.

Remarks

SetPlatformPathOverride is used to change the source path when queuing files. If a platform path override has been set by a call to **SetPlatformPathOverride**, any setup function that queues file copy operations will examine the final component of the source path and if the final component matches the name of the user's platform, replace it with the override string set by **SetPlatformPathOverride**.

For example, consider a MIPS-platform machine where the platform has been set to Alpha by a call to **SetPlatformPathOverride**. After the platform path override has been set, a file copy operation is queued with a source path of \\foo\bar\baz\mips\x.exe, the path will be changed to \\foo\bar\baz\alpha\x.exe.

The paths of file copy operations queued before the path override is set are not changed.

See Also

SetupSetDirectoryId

SetupSetSourceList Group

Group

Group

[New - Windows NT]

The **SetupSetSourceList** function allows the caller to set the list of installation sources for either the current user or the system (common to all users).

```
BOOL SetupSetSourceList(
    DWORD Flags,            // type of source list
    PCTSTR *SourceList,     // array of sources listed
    UINT SourceCount        // number of sources in the array
  );
```

Parameters

Flags

These flags specify the type of list. This parameter can be a combination of the following values:

SRCLIST_SYSTEM

The list is the per-system Most Recently Used (MRU) list stored in the registry. The caller must be a member of the administrators local group.

SRCLIST_USER

The list is the per-user MRU list stored in the registry.

SRCLIST_TEMPORARY

The specified list is temporary and will be the only list accessible to the current process until **SetupCancelTemporarySourceList** is called or **SetSourceList** is called again.

Important

If a temporary list is set, sources will not be added to or deleted from the system or user lists, even if subsequent calls to **SetupAddToSourceList** or **SetupRemoveFromSourceList** explicitly specify those lists.

Note One of the SRCLIST_SYSTEM, SRCLIST_USER, or SRCLIST_TEMPORARY flags must be specified. SRCLIST_NOBROWSE

The user is not allowed to add or change sources when **SetupPromptForDisk** is used. This flag is typically used in combination with the SRCLIST_TEMPORARY flag.

SourceList

Pointer to an array of strings to use as the source list, as specified by the *Flags* parameter.

SourceCount

Specifies the number of elements in the array pointed to by *SourceList*.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

See Also

SetupAddToSourceList, **SetupCancelTemporarySourceList**, **SetupRemoveFromSourceList**

SetupTermDefaultQueueCallback Group

Group

Group

[New - Windows NT]

The **SetupTermDefaultQueueCallback** function is called after a queue has finished committing. It frees resources allocated by previous calls to **SetupInitDefaultQueueCallback** or **SetupInitDefaultQueueCallbackEx**.

```
VOID SetupTermDefaultQueueCallback(
    PVOID Context       // context used by the default callback routine
  );
```

Parameters

Context

Supplies a pointer to the context used by the default callback routine.

Return Values

Does not return a value.

See Also

SetupInitDefaultQueueCallback, **SetupInitDefaultQueueCallbackEx**

SetupTerminateFileLog Group

Group

Group

[New - Windows NT]

The **SetupTerminateFileLog** function releases resources associated with a file log.

```
BOOL SetupTerminateFileLog(
    HSPFILELOG FileLogHandle      // handle of the log file to close
  );
```

Parameter

FileLogHandle

Supplies the handle to the log file as returned by a call to **SetupInitializeFileLog**.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

See Also

SetupLogFile, **SetupInitializeFileLog**

Notifications

Notifications are values that a setup function sends to a callback routine to specify a state or event. Two parameters, *Param1* and *Param2*, are sent with the notification, and contain additional information relevant to the notification.

The callback routine processes the notification and returns an unsigned integer to the setup function. Depending on the setup function, you can use this value may be used to specify an operation or user selection, or you may ignore it.

The setup functions send notifications to callback routines using the following syntax.

```
MsgHandler(          //the specified callback routine
    Context,         //context used by the callback routine
    Notification,    //notification code
    Param1,          //additional notification information
    Param2            //additional notification information
);
```

The *Context* parameter is a void pointer to a context variable or structure that the callback routine can use to store information that must persist between subsequent calls to the callback routine.

Because the callback routine specifies the context's implementation, and it is never referenced or altered by the setup functions, the context is not documented in the reference material for the notification messages that follow.

The *Notification* parameter specifies an unsigned integer value for an event or state that causes the setup function to call the callback routine.

Param1 and *Param2* are optional parameters that can contain additional information relevant to the notification. These parameters are unsigned integers. If *Param1* or *Param2* return information that is not an unsigned integer, it will be cast to an unsigned integer and must be recast to its original data type before it can be used by the callback routine.

Note The following notifications represent every notification used by the setup functions. Individual functions will use a subset of these notifications. In other words, not every notification is used by every function.

The following notifications are used by the setup functions.

SPFILENOTIFY_COPYERROR	An error occurred during a file copying operation.
SPFILENOTIFY_DELETEERROR	An error occurred during a file deletion operation.
SPFILENOTIFY_ENDCOPY	A file copying operation has ended.
SPFILENOTIFY_ENDDELETE	A file deletion operation has ended.
SPFILENOTIFY_ENDQUEUE	The queue has finished committing.
SPFILENOTIFY_ENDRENAME	A file rename operation has ended.
SPFILENOTIFY_ENDSUBQUEUE	A subqueue (either copy,

	rename or delete) has ended.
SPFILENOTIFY_FILEEXTRACTED	The file has been extracted from the cabinet.
SPFILENOTIFY_FILEINCABINET	A file is encountered in the cabinet.
SPFILENOTIFY_FILEOPDELAYED	The file was in use, and the current operation has been delayed until the system is rebooted.
SPFILENOTIFY_LANGMISMATCH	The language of the current operation does not match the system language.
SPFILENOTIFY_NEEDMEDIA	New source media is needed.
SPFILENOTIFY_NEEDNEWCABINET	The current file is continued in the next cabinet.
SPFILENOTIFY_QUEUESCAN	A node in the file queue has been scanned.
SPFILENOTIFY_RENAMEERROR	An error occurred during a file renaming operation.
SPFILENOTIFY_STARTCOPY	A file copy operation has started.
SPFILENOTIFY_STARTDELETE	A file delete operation has started.
SPFILENOTIFY_STARTQUEUE	The queue has started to commit.
SPFILENOTIFY_STARTRENAME	A file rename operation has started.
SPFILENOTIFY_STARTSUBQUEUE	A subqueue (either copy, rename or delete) has started.
SPFILENOTIFY_TARGETEXISTS	A copy of the specified file already exists on the target.
SPFILENOTIFY_TARGETNEWER	A newer version of the specified file exists on the target.

SPFILENOTIFY_COPYERROR Group

Group

Group

[New - Windows NT]

The SPFILENOTIFY_COPYERROR notification is sent to the callback routine if an error occurs during a file copy operation.

```
SPFILENOTIFY_COPYERROR
    Param1 = (UINT) FilePathInfo;
    Param2 = (UINT) ReturnBuffer;
```

Parameters

Param1

Pointer to a **FILEPATHS** structure.

Param2

Pointer to a buffer, of size MAX_PATH TCHAR elements, that stores new path information specified by the user.

Return Values

The callback should return one of the following values.

Value	**Meaning**
FILEOP_ABORT	Queue processing should be cancelled. **SetupCommitFileQueue** returns FALSE and **GetLastError** returns extended error information such as ERROR_CANCELLED (if the user canceled) or ERROR_NOT_ENOUGH_MEMORY.
FILEOP_NEWPATH	Retry the copy operation using the path the callback function placed in the buffer pointed to by the *Param2* parameter. The callback routine should ensure that the path does not overflow the buffer size of a TCHAR array of MAX_PATH elements.
FILEOP_RETRY	The user chose to attempt the the copy operation again.
FILEOP_SKIP	The user chose to skip the file copy operation.

See Also

FILEPATHS, **SetupCommitFileQueue**, **SetupDefaultQueueCallback**

SPFILENOTIFY_DELETEERROR

Group

Group

Group

[New - Windows NT]

The SPFILENOTIFY_DELETEERROR notification is sent to the callback routine if an error occurs during a file delete operation.

```
SPFILENOTIFY_DELETEERROR
    Param1 = (UINT) FilePathInfo;
    Param2 = (UINT) 0;
```

Parameters

Param1
Pointer to a **FILEPATHS** structure.

Param2
Unused.

Return Values

The callback routine should return one of the following values.

Value	**Meaning**
FILEOP_ABORT	Queue processing should be canceled. **SetupCommitFileQueue** returns FALSE and **GetLastError** returns extended error information such as ERROR_CANCELLED (if the user canceled) or ERROR_NOT_ENOUGH_MEMORY.
FILEOP_RETRY	The user chose to attempt the delete operation again.
FILEOP_SKIP	The user chose to skip the file delete operation.

See Also

FILEPATHS, **SetupCommitFileQueue**, **SetupDefaultQueueCallback**

SPFILENOTIFY_ENDCOPY Group

Group

Group

[New - Windows NT]

The SPFILENOTIFY_ENDCOPY notification is passed to the callback function when the queue completes a copy operation. This notification is sent even if the user cancels or if an error occurs.

```
SPFILENOTIFY_ENDCOPY
    Param1 = (UINT) FilePathInfo;
    Param2 = (UINT) 0;
```

Parameters

Param1

Pointer to a **FILEPATHS** structure. The **Win32Error** member of the **FILEPATHS** structure indicates the outcome of the copy operation.

Param2

Unused.

Return Values

The return code is ignored.

See Also

FILEPATHS, **SetupCommitFileQueue**, **SetupDefaultQueueCallback**

SPFILENOTIFY_ENDDELETE

Group

Group

Group

[New - Windows NT]

The SPFILENOTIFY_ENDDELETE notification is returned to the callback routine when a queue completes a delete operation. This notification is sent even if the user cancels or if an error occurs.

```
SPFILENOTIFY_ENDDELETE
    Param1 = (UINT) FilePathInfo;
    Param2 = (UINT) 0;
```

Parameters

Param1

Pointer to a FILEPATHS structure. The **Win32Error** member of the **FILEPATHS** structure indicates the outcome of a copy operation.

Param2

Unused.

Return Values

The return code is ignored.

See Also

FILEPATHS, SetupCommitFileQueue, SetupDefaultQueueCallback

SPFILENOTIFY_ENDQUEUE Group

Group

Group

[New - Windows NT]

The SPFILENOTIFY_ENDQUEUE notification is sent to the callback routine when all of the queued operations have been completed.

```
SPFILENOTIFY_ENDQUEUE
    Param1 = (UINT) Result;
    Param2 = (UINT) 0;
```

Parameters

Param1
: TRUE if the queue was processed successfully, FALSE otherwise.

Param2
: Unused.

Return Values

The return value is ignored.

See Also

SetupCommitFileQueue, **SetupDefaultQueueCallback**

SPFILENOTIFY_ENDRENAME Group

Group

Group

[New - Windows NT]

The SPFILENOTIFY_ENDRENAME notification is sent to the callback routine when the queue completes a rename operation. This notification is sent even if the user cancels or if an error occurs.

```
SPFILENOTIFY_ENDRENAME
    Param1 = (UINT) FilePathInfo;
    Param2 = (UINT) 0;
```

Parameters

Param1

Pointer to a FILEPATHS structure. The **Win32Error** member of the **FILEPATHS** structure indicates the outcome of the copy operation.

Param2

Unused.

Return Values

The return value is ignored.

See Also

FILEPATHS, **SetupCommitFileQueue**, **SetupDefaultQueueCallback**

SPFILENOTIFY_ENDSUBQUEUE Group

Group

Group

[New - Windows NT]

The SPFILENOTIFY_ENDSUBQUEUE notification is sent to the callback function when the queue completes all the operations in the delete, rename, or copy subqueue.

```
SPFILENOTIFY_ENDSUBQUEUE
    Param1 = (UINT) SubQueue;
    Param2 = (UINT) 0;
```

Parameters

Param1

Specifies which subqueue has been completed. This parameter can be one of the following values FILEOP_DELETE, FILEOP_RENAME, or FILEOP_COPY.

Param2

Unused.

Return Values

The return value is ignored.

See Also

SetupCommitFileQueue, **SetupDefaultQueueCallback**

SPFILENOTIFY_FILEEXTRACTED Group

Group

Group

[New - Windows NT]

The SPFILENOTIFY_FILEEXTRACTED notification is sent to a callback routine by **SetupIterateCabinet** to indicate either that a file was extracted from the cabinet or that an extraction failed and cabinet processing has been canceled.

```
SPFILENOTIFY_FILEEXTRACTED
    Param1 = (UINT) FilePathInfo;
    Param2 = (UINT) 0;
```

Parameters

Param1

Pointer to a **FILEPATHS** structure that contains path information for the extracted file. The **SourceFile** member of the **FILEPATHS** structure contains the full Win32 source path of the cabinet. The **TargetFile** member supplies the full Win32 target path of the file to be installed on the system.

Param2

Unused.

Return Values

The cabinet callback routine should return one of the following values.

Value	**Meaning**
NO_ERROR	No error was encountered, continue processing the cabinet.
ERROR_XXX	An error of the specified type occurred. **SetupIterateCabinet** will return FALSE. **GetLastError** will return the specified error code.

Note There is no default cabinet callback routine supplied with the Setup API. Your setup application should supply a callback routine to handle the notifications sent by the **SetupIterateCabinet** function.

See Also

FILEPATHS, **SetupIterateCabinet**

SPFILENOTIFY_FILEINCABINET Group

Group

Group

[New - Windows NT]

The SPFILENOTIFY_FILEINCABINET notification is sent to a callback routine by **SetupIterateCabinet** for each file found in the cabinet. The callback routine must return a value indicating whether to extract the file.

```
SPFILENOTIFY_FILEINCABINET
    Param1 = (UINT) FileInCabinetInfo;
    Param2 = (UINT) CabinetFile;
```

Parameters

Param1
Specifies the address to a **FILE_IN_CABINET_INFO** structure that contains information about the file in the cabinet.

Param2
Pointer to a null-terminated string that contains the filename of the cabinet file.

Return Values

Your callback routine should return one of the following.

Value	**Meaning**
FILEOP_SKIP	Do not extract the file, skip it.
FILEOP_DOIT	Extract the file.

If your callback routine returns FILEOP_DOIT, the name to use for the extracted file should be specified in the **FullTargetName** member of the **FILE_IN_CABINET_INFO** structure passed to the routine in *Param1*.

Note There is no default cabinet callback routine. The setup application should supply a callback routine to handle the notifications sent by **SetupIterateCabinet**.

See Also

FILE_IN_CABINET_INFO,
SetupIterateCabinet

SPFILENOTIFY_FILEOPDELAYED Group

Group

Group

[New - Windows NT]

The SPFILENOTIFY_FILEOPDELAYED notification is sent by **SetupInstallFileEx** or **SetupCommitFileQueue** to a callback routine when a file operation was delayed because the file was in use. The operation will be processed the next time the system is rebooted.

```
SPFILENOTIFY_FILEOPDELAYED
    Param1 = (UINT) FilePathInfo;
    Param2 = (UINT) 0;
```

Parameters

Param1

Pointer to a **FILEPATHS** structure.

If the delayed operation is a file copy operation, the **FILEPATHS** structure contains the following information.

FILEPATH member	Value
Win32Error	NO_ERROR
Flags	FILEOP_COPY
Source	The full Win32 path of the temporary file. This temporary file will be copied to the target directory when the system is rebooted. The setup functions automatically generate a path for the temporary file.
Target	Specifies the full Win32 path of the actual target file.

If the delayed operation is a file delete operation, the **FILEPATHS** structure contains the following information.

FILEPATH member	Value
Win32Error	NO_ERROR
Flags	FILEOP_DELETE
Source	NULL
Target	Specifies the full Win32 path of the file to be deleted.

Param2

Is not used.

Return Values

The return value is ignored.

See Also

FILEPATHS, **SetupCommitFileQueue**, **SetupInstallFile**, **SetupInstallFileEx**,

SetupInstallFromInfSection

SPFILENOTIFY_LANGMISMATCH Group

Group

Group

[New - Windows NT]

The SPFILENOTIFY_LANGMISMATCH notification is sent to the callback routine if the language of the file to be copied does not match the language of an existing target file. It can be sent to the callback routine alone or combined, by using the OR operator, with SPFILENOTIFY_TARGETEXISTS and/or SPFILENOTIFY_TARGETNEWER.

```
SPFILENOTIFY_LANGMISMATCH
    Param1 = (UINT) FilePathInfo;
    Param2 = (UINT) LanguageInfo;
```

Parameters

Param1

Pointer to a **FILEPATHS** structure that contains information about the paths of the source and target files.

Param2

An unsigned 32-bit value that contains information about the mis-matched languages. This **DWORD** stores the language identifier of the source file in the low word, and the language identifier of the existing target file in the high word.

Return Values

The callback routine should return one of the following values.

Value	Meaning
TRUE	Copy the file and overwrite the existing file.
FALSE	Skip the copy operation. Do not overwrite the existing file.

See Also

FILEPATHS, **SetupCommitFileQueue**, **SetupDefaultQueueCallback**, **SetupInstallFile**, **SetupInstallFileEx**, **SetupInstallFromInfSection**

SPFILENOTIFY_NEEDMEDIA Group

Group

Group

[New - Windows NT]

The SPFILENOTIFY_NEEDMEDIA notification is sent to the callback routine when new media or a new cabinet file is required.

```
SPFILENOTIFY_NEEDMEDIA
    Param1 = (UINT) SourceMediaInfo;
    Param2 = (UINT) NewPathInfo;
```

Parameters

Param1

Pointer to a **SOURCE_MEDIA** structure.

Param2

Pointer to a character array to store new path information supplied by the user. The buffer size is a TCHAR array of MAX_PATH elements. The path information returned by your setup application should not exceed this size.

Return Values

The callback routine should return one of the following.

Value	**Meaning**
FILEOP_NEWPATH	The media is present and the requested file is available at the win32 path specified in the buffer pointed to by *Param2*.
FILEOP_SKIP	Skip the requested file
FILEOP_ABORT	Abort queue processing. The **SetupCommitFileQueue** function returns FALSE. GetLastError returns extended error information, such as ERROR_CANCELLED if the user canceled.
FILEOP-DOIT	The media is available.

See Also

SetupCommitFileQueue, **SetupDefaultQueueCallback**, **SOURCE_MEDIA**

SPFILENOTIFY_NEEDNEWCABINET Group

Group

Group

[New - Windows NT]

The SPFILENOTIFY_NEEDNEWCABINET notification is sent by **SetupIterateCabinet** to indicate that the current file continues in another cabinet. Your callback routine can then call **SetupPromptForDisk**, or create its own dialog box to prompt the user to insert the next disk.

```
SPFILENOTIFY_NEEDNEWCABINET
    Param1 = (UINT) CabinetInfo;
    Param2 = (UINT) NewPath;
```

Parameters

Param1

Pointer to a **CABINET_INFO** structure that contains information about the cabinet and the file to be extracted.

Param2

If the callback returns NO_ERROR, this parameter is a pointer to a null-terminated string. If the string is not empty, it specifies a new path to the cabinet.

Return Values

Your routine should return one of the following values.

Value	Meaning
NO_ERROR	No error was encountered, continue processing the cabinet.
ERROR_XXX	An error of the specified type occurred. The **SetupIterateCabinet** function will return FALSE, and the specified error code will be returned by a call to **GetLastError**.

Note There is no default cabinet callback routine; thus, you must supply a callback routine to handle the notifications sent by **SetupIterateCabinet**.

Remarks

If the callback routine returns NO_ERROR, **SetupIterateCabinet** checks the buffer pointed to by *Param2*. If the buffer is not empty, then it contains a new source path. If the buffer is empty, the source path is assumed to be unchanged.

Your callback function should ensure that the cabinet is accessible before it returns, calling the **SetupPromptForDisk** function, if new media needs to be inserted.

See Also

CABINET_INFO, **SetupIterateCabinet**

SPFILENOTIFY_QUEUESCAN Group

Group

Group

[New - Windows NT]

The SPFILENOTIFY_QUEUESCAN notification is sent to a callback routine by **SetupScanFileQueue** for each node in the copy subqueue of the file queue. This only occurs if the **SetupScanFileQueue** function was called specifiying the flag SPQ_SCAN_USE_CALLBACK.

```
SPFILENOTIFY_QUEUESCAN
    Param1 = (UINT) TargetPath;
    Param2 = (UINT) DelayFlag;
```

Parameters

Param1

Specifies a null-terminated string that specifies the target path information for the file queued in the current node.

Param2

If the file in the current node of the queue is in use, *Param2* takes the value SPQ_DELAYED_COPY. If the file is not in use, the value is zero.

Return Values

The callback routine should return a Win32 error code.

If the callback routine returns NO_ERROR, the queue scan continues. If the routine returns any other Win32 error code, the queue scan aborts and **SetupScanFileQueue** returns FALSE.

Note This notification is not handled by the **SetupDefaultQueueCallback** function.

See Also

SetupScanFileQueue

SPFILENOTIFY_RENAMEERROR Group

Group

Group

[New - Windows NT]

The SPFILENOTIFY_RENAMEERROR notification is sent to the callback routine if an error occurs during a file rename operation.

```
SPFILENOTIFY_RENAMEERROR
    Param1 = (UINT) FilePathInfo;
    Param2 = (UINT) 0;
```

Parameters

Param1

Pointer to a **FILEPATHS** structure. The **Win32Error** member of the **FILEPATHS** structure specifies the error that occurred during the rename operation.

Param2

Unused.

Return Values

The callback routine should return one of the following.

Value	**Meaning**
FILEOP_RETRY	The user chose to attempt the rename operation again.
FILEOP_SKIP	The user chose to skip the file rename operation.
FILEOP_ABORT	Stop the queue commit. **SetupCommitFileQueue** returns FALSE. **GetLastError** returns extended error information such as ERROR_CANCELLED (if the user canceled) or ERROR_NOT_ENOUGH_MEMORY.

See Also

FILEPATHS, **SetupCommitFileQueue**, **SetupDefaultQueueCallback**

SPFILENOTIFY_STARTCOPY Group

Group

Group

[New - Windows NT]

The SPFILENOTIFY_STARTCOPY notification is sent to the callback function when the queue starts a file copy operation.

```
SPFILENOTIFY_STARTCOPY
    Param1 = (UINT) FilePathInfo;
    Param2 = (UINT) Operation;
```

Parameters

Param1

Pointer to a **FILEPATHS** structure.

Param2

Always has the value FILEOP_COPY.

Return Values

The callback routine should return one of the following values.

Value	Meaning
FILEOP_ABORT	Abort the queue commit process. The callback routine should call **SetLastError** to indicate the reason for termination. The **SetupCommitFileQueue** function returns FALSE and a subsequent call to **GetLastError** returns the error code set by the callback routine during the call to **SetLastError**.
FILEOP_DOIT	Perform the file copy operation.
FILEOP_SKIP	Skip the current copy operation.

See Also

FILEPATHS, **SetupCommitFileQueue**, **SetupDefaultQueueCallback**

SPFILENOTIFY_STARTDELETE Group

Group

Group

[New - Windows NT]

The SPFILENOTIFY_STARTDELETE notification is sent to the callback function when the queue starts a file delete operation.

```
SPFILENOTIFY_STARTDELETE
    Param1 = (UINT) FilePathInfo;
    Param2 = (UINT) Operation;
```

Parameters

Param1

Pointer to a **FILEPATHS** structure.

Param2

Always has the value FILEOP_DELETE.

Return Values

The callback routine should return one of the following values.

Value	**Meaning**
FILEOP_ABORT	Abort the queue commit. The callback routine should call **SetLastError** to indicate the reason for aborting. **SetupCommitFileQueue** returns FALSE and a subsequent call to **GetLastError** returns the error code set by the callback routine during the call to **SetLastError**.
FILEOP_DOIT	Perform the file copy operation.
FILEOP_SKIP	Skip the current copy operation.

See Also

FILEPATHS, **SetupCommitFileQueue**, **SetupDefaultQueueCallback**

SPFILENOTIFY_STARTQUEUE Group

Group

Group

[New - Windows NT]

The SPFILENOTIFY_STARTQUEUE notification is sent to the callback routine when the queue commitment process starts.

```
SPFILENOTIFY_STARTQUEUE
    Param1 = (UINT) OwnerWindow;
    Param2 = (UINT) 0;
```

Parameters

Param1

Handle to the window that is to own the dialog boxes that the callback routine generates.

Param2

Unused.

Return Values

If an error occurs, the callback routine should call **SetLastError**, specifying the error, and then return 0. The **SetupCommitFileQueue** function will return FALSE and a subsequent call to **GetLastError** will return the error code set by the callback routine.

If no error occurs, the callback routine should return a nonzero value.

See Also

SetupCommitFileQueue, **SetupDefaultQueueCallback**

SPFILENOTIFY_STARTRENAME Group

Group

Group

[New - Windows NT]

The SPFILENOTIFY_STARTRENAME notification is sent to the callback function when the queue starts a file rename operation.

```
SPFILENOTIFY_STARTRENAME
    Param1 = (UINT) FilePathInfo;
    Param2 = (UINT) Operation;
```

Parameters

Param1
Pointer to a **FILEPATHS** structure.

Param2
Always has the value FILEOP_RENAME.

Return Values

The callback routine should return one of the following values.

Value	**Meaning**
FILEOP_ABORT	Abort the queue commit. The callback routine should call **SetLastError** to indicate the reason for termination. **SetupCommitFileQueue** returns FALSE and a subsequent call to **GetLastError** returns the error code set by the callback routine during the call to **SetLastError**.
FILEOP_DOIT	Perform the file copy operation.
FILEOP_SKIP	Skip the current copy operation.

See Also

FILEPATHS, **SetupCommitFileQueue**, **SetupDefaultQueueCallback**

SPFILENOTIFY_STARTSUBQUEUE Group

Group

Group

[New - Windows NT]

The SPFILENOTIFY_STARTSUBQUEUE notification is sent to the callback function when the queue starts to process the operations in the delete, rename, or copy subqueue.

```
SPFILENOTIFY_STARTSUBQUEUE
    Param1 = (UINT) SubQueue;
    Param2 = (UINT) NumOperations;
```

Parameters

Param1
Specifies which subqueue has been started. This parameter can be any one of the values FILEOP_DELETE, FILEOP_RENAME, or FILEOP_COPY.

Param2
Specifies the number of file copy, rename, or delete operations in the subqueue.

Return Values

If an error occurs, the callback routine should call **SetLastError**, specifying the error, and then return 0. The **SetupCommitFileQueue** function will return FALSE and a subsequent call to **GetLastError** will return the error code set by the callback routine.

If no error occurs, the callback routine should return a nonzero value.

See Also

SetupCommitFileQueue, **SetupDefaultQueueCallback**

SPFILENOTIFY_TARGETEXISTS

Group

Group

Group

[New - Windows NT]

The SPFILENOTIFY_TARGETEXISTS notification is sent to the callback routine if the file to be copied was queued with the SP_COPY_NOOVERWRITE flag and that file already exists in the target directory. It can be sent to the callback routine alone or combined, by using the OR operator, with the SPFILENOTIFY_LANGMISMATCH and/or SPFILENOTIFY_TARGETNEWER notifications.

```
SPFILENOTIFY_TARGETEXISTS
    Param1 = (UINT) FilePathInfo;
    Param2 = (UINT) 0;
```

Parameters

Param1

Pointer to a **FILEPATHS** structure that contains information about the paths for the source and target files.

Param2

This parameter is not used unless this notification is combined, by using the OR operator, with the SPFILENOTIFY_LANGMISMATCH notification.

Return Values

The callback routine should return one of the following values.

Value	Meaning
TRUE	Overwrite the file in the target directory.
FALSE	Skip the current copy operation.

See Also

FILEPATHS, **SetupCommitFileQueue**, **SetupDefaultQueueCallback**, **SetupInstallFile**, **SetupInstallFileEx**, **SetupInstallFromInfSection**

SPFILENOTIFY_TARGETNEWER Group

Group

Group

[New - Windows NT]

The SPFILENOTIFY_TARGETNEWER notification is sent to the callback routine if the file to be copied was queued with the SP_COPY_NEWER or SP_COPY_FORCE_NEWER flags specified and a newer version of the file already exists in the target directory. It can be sent to the callback routine alone or ORed together with SPFILENOTIFY_LANGMISMATCH and/or SPFILENOTIFY_TARGETEXISTS.

```
SPFILENOTIFY_TARGETNEWER
    Param1 = (UINT) FilePathInfo;
    Param2 = (UINT) 0;
```

Parameters

Param1

Pointer to a **FILEPATHS** structure that contains information about the paths for source and target files.

Param2

This parameter is not used unless this notification is combined, by using the OR operator, with SPFILENOTIFY_LANGMISMATCH.

Return Values

The callback routine should return one of the following values.

Value	**Meaning**
TRUE	Overwrite the file in the target directory.
FALSE	Skip the current copy operation.

See Also

FILEPATHS, **SetupCommitFileQueue**, **SetupDefaultQueueCallback**, **SetupInstallFile**, **SetupInstallFileEx**, **SetupInstallFromInfSection**

Errors

The following errors are specific to the Setup API.

Error Codes

The following error codes are specific to the Setup API.

INF Parsing Errors	**Description**
ERROR_EXPECTED_SECTION_NAME	A section name was expected, and not found.
ERROR_BAD_SECTION_NAME_LINE	The section name was not of the correct format. (For example, a name not terminated by a right-hand bracket ( ]).
ERROR_SECTION_NAME_TOO_LONG	The section name exceeded the maximum length of MAX_SECT_NAME_LEN.
ERROR_GENERAL_SYNTAX	The general syntax is incorrect.

INF Runtime Errors	**Description**
ERROR_WRONG_INF_STYLE	The INF is not of the type specified in the function call. (This error will also be returned if an Windows NT 3.*x* INF file is passed into **SetupOpenAppendInfFile**).
ERROR_SECTION_NOT_FOUND	The section was not found in the INF file.
ERROR_LINE_NOT_FOUND	The line was not found in the INF section.