WHITE PAPER

# Designing an Update-Friendly MSI Installation

by Robert Dickau
Principal Engineer, Flexera Software

FLEXERA SOFTWARE

# Designing an Update-Friendly MSI Installation

## Introduction

Creating and deploying software updates is standard procedure for virtually every software company in the world. Knowing strategies for how to create an update-friendly Windows Installer (MSI) installation goes a long way to ensuring a smooth, error-free update experience for your end users down the road.

In this white paper, you will learn about designing your original Windows Installer setup project to best prepare it for future upgrades, and how to design upgrade packages to install later versions of your products. It will also provide an introduction to the different types of updates supported by Windows Installer. Finally, at times throughout the white paper it will explain how InstallShield® from Flexera Software can assist with the installation and update authoring process.

## Types of Upgrades

Windows Installer supports three types of product upgrades: small updates, minor upgrades, and major upgrades. The three types of upgrades are defined as follows.

- A small update consists of product changes, such as hot fixes, so small that no change to the product version is necessary or desired. (A drawback to small updates is that external programs, including installers for later versions of your product, will not be able to distinguish a product with the small update applied from one without the small update.)
- A minor upgrade is a change to the product large enough to merit a change to the product version, such as updating version 1.1 to 1.2, but in which there have been no significant changes to the setup organization between versions. The install-time behavior of a minor upgrade is to install over the existing product.
- A major upgrade includes substantial product changes, such as updating version 1.2 to 2.0. A major upgrade can contain significant changes to the setup architecture. The install-time behavior of a major upgrade can be to uninstall the earlier version and install the new one, or to install over the earlier version and then remove any leftover data.

*NOTE: For an earlier product version that was installed with a legacy (non-MSI) setup, a custom action will normally be required to uninstall or modify the existing product installation.*

## Packaging and Deploying Upgrades

Windows Installer provides different methods for packaging upgrades, and the different options affect the way the upgrade is applied to a target system.

### Packaging Options

An upgrade can be packaged for deployment to the target system as a full installation (MSI package). An upgrade packaged as a full installation can be authored (using custom actions, command-line switches, or a setup launcher) to upgrade an existing product if one is present, or otherwise to behave as a first-time installation.

An upgrade can also be packaged as a Windows Installer patch file (a file with the MSP extension). A Windows Installer patch contains changes between the files (and other data) and MSI tables in the earlier and later versions. The file differences stored in a patch can be binary, byte-level differences, possibly resulting in a much smaller deliverable than an update packaged as a full installation package. An update that you package as a patch file can be used only to upgrade an existing, installed product, and cannot be used as a first-time installation.

Small updates and minor upgrades are commonly packaged as patches, while major upgrades are usually packaged as full installation packages.

*NOTE: A common misapprehension is that patches are a separate type of upgrade, as opposed to a packaging mechanism. In fact, the patch-development process involves first designing a minor or major upgrade, and then packaging it as a patch. Before creating a patch, it is recommended you test the update as a full installation package.*

### Deploying Upgrades and Patches

When you run an MSI installation package for the first time on a given system, Windows Installer caches the MSI database in the hidden directory %WINDIR%\Installer. By default, when you run the same package a second or later time on the same system, Windows Installer runs the package in *maintenance mode*, using the cached database. (A package is typically authored to show a different series of dialog boxes for first-time installations and maintenance-mode installations, using conditions such as "Not Installed".)

During the initial installation, the MSI database is cached on the user's system, but the product's data files are not. If a maintenance operation results in a file having to be installed, MSI will require access to the original installation source, prompting the user to locate the source if it cannot be found (for example, if the installation was performed from a DVD that is no longer in the drive). For this reason, you should either build a release with the MSI database external to a setup launcher, or create a setup launcher that caches the installation on the local machine.

When you deploy a *major upgrade* package, no special command-line switches or property values are required. When deploying a *minor upgrade* package, however, you will generally need to set appropriate values for the REINSTALLMODE and REINSTALL properties, as described in the following section.

### About REINSTALLMODE and REINSTALL

To avoid maintenance mode for a small update or minor upgrade installer, the MSI property REINSTALLMODE must be set at the command line, either by the user or by a setup launcher. The REINSTALLMODE property defines what types of data should be reinstalled: the value is a string of characters, where each character indicates a particular type of data to reinstall. (A major upgrade typically does not need any special properties set at the command line.)

The default REINSTALLMODE value is "omus", where the characters stand for the following:

- *o*: reinstall a file only if it is missing from the target system, or if the existing file on the target system is older.
- *m*: reinstall machine-wide registry data.
- *u*: reinstall user-specific registry data.
- *s*: reinstall shortcuts.

When deploying a small update or minor upgrade, the key is to re-cache the cached MSI database by including the letter "v" in the REINSTALLMODE value, as in REINSTALLMODE=voums. (The order of characters in the REINSTALLMODE value is unimportant.)

*NOTE: The "v" option for REINSTALLMODE must be set at the command line when the minor upgrade installation is launched; the other REINSTALLMODE settings can be activated within a running installation. InstallShield can help you create a setup launcher for a minor upgrade that detects if an earlier version of a product is installed on a system, and sets REINSTALLMODE and REINSTALL appropriately. Moreover, MSI validation rule ICE40 posts a warning if REINSTALLMODE is set in the Property table.*

For an update installer, the REINSTALL property should also typically be set. The REINSTALL property should be set to a comma-separated list of features to reinstall (using the internal feature names, and not the localized display names), or to the special value "ALL". Setting REINSTALL to ALL causes only the features already installed by an earlier installation to be reinstalled. For this reason, setting REINSTALL to ALL is inappropriate for a first-time installation: during a first-time installation, no features have yet been installed.

When running a minor upgrade packaged as a full MSI package, a typical command line is the following:

```
msiexec /i ProductName.msi REINSTALLMODE=voums
REINSTALL=ALL
```

A patch can be distributed using the MSP file, or by creating an Update.exe file that wraps the MSP and passes the appropriate REINSTALLMODE and REINSTALL property values to the Windows Installer engine.

To deploy a patch, a typical command line is the following:

```
msiexec /p patch.msp REINSTALLMODE=oums
REINSTALL=ALL
```

Because a patch does not modify the existing cached MSI database, including the "v" setting for REINSTALLMODE is unnecessary.

At run time, a patch transforms the cached MSI database, and then runs it in maintenance mode. A patch file is also cached on a target system, in the same location as cached MSI databases.

## Designing an Update-Friendly Installation

The design of your installation projects—the organization of a product's features, components, and key paths—has an impact on the effectiveness of future updates. The effectiveness of an upgrade or patch is gauged by the following:

- The new package updates the appropriate installed product: the package installs new and updated product data, and does not remove any required data.
- An update packaged as a full installation should behave correctly as a first-time installation if no earlier version exists on the target system. (An update packaged as a patch cannot act as a first-time installation.)
- The product information registered on the system should display information only for the newest product version. There should not, for example, be more than one entry for your product present in the Programs and Features panel.

For patch packages, there are additional considerations:

- The patch package should be as small as possible, when appropriate containing byte-level differences between the files in your earlier and later installation packages.
- The patch should avoid unnecessary prompts for the source media.

This white paper provides some general guidelines for creating update-friendly projects, both for the original installation and for update installations.

## Organizing the Original Project

The design of the first release of your original installation project can have a significant effect on the success of later updates applied to it. This section offers some tips for organizing your initial MSI installation project, and where appropriate describes the applicable Windows Installer behavior or best-practices guidelines that motivate these tips.

*NOTE: These tips apply largely to minor upgrades. In general, the uninstall-then-reinstall nature of major upgrades makes them less susceptible to problems related to the organization of an installation project. The InstallShield Upgrade Validation Wizard automates detection that many of these rules have been followed. To launch the wizard, use the Build > Validate > Upgrade Validation Wizard command. " The InstallShield help topic "Validators" describes the tests performed during upgrade validation.*

**Tip 1: Whenever possible, use versioned key files.**

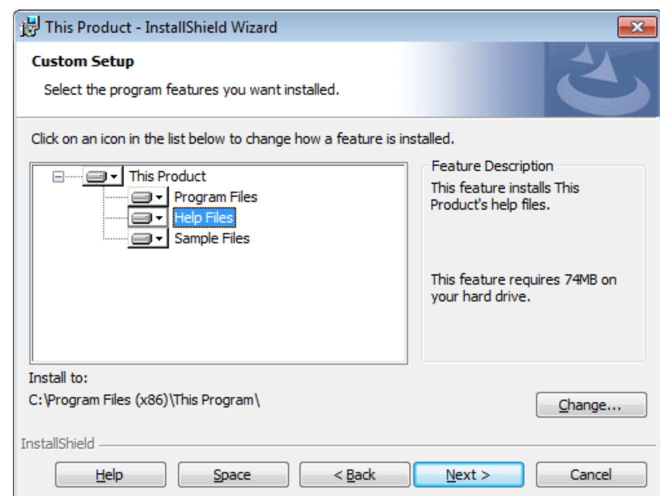
As described later in this white paper, part of Windows Installer's contribution to system stability is the enforcement of strict file-versioning rules. However, MSI ordinarily performs version comparison only on the *key file* of a component when deciding whether to install a component during a reinstallation or update installation. A simple way to ensure that a component will be updated in your new product version is to give your key file a newer version than the corresponding file on the target system.

This tip is related to the MSI best-practice rule of putting at most one portable executable file—EXE, DLL, OCX, and the like—in a component, and marking that file as the key file of its component. In addition to giving you the most effective repair mode for your installation, having more components leads to more desirable default behavior if only some of your files are updated in your new product version.

**Tip 2: Partition your product into discrete sets of features.**

The primary use of features is to provide user-selectable pieces of your product's functionality. An early part of the design of your installation program is to define the features (and subfeatures, and so forth) that you want the user to be able to see and configure.

Most installers provide a custom setup type, which displays to the user a panel similar to the following, where the user can select which product features to install.




This end-user view of your installer is the foundation of the other features you need to configure.

There is no fixed list of rules for dividing an application into features. In some cases, the architecture of your application will suggest divisions into features (Program Files, Help Files, Tools, Examples, and so forth). In other cases, it will be necessary to define artificial boundaries within your application to create features of a manageable size.

After you partition your product into user-selectable features, you can further divide these features into subfeatures. For each of these subfeatures, you can set the Display attribute to Not Visible, set the Remote Installation attribute to Favor Parent, and set the Required attribute to Yes. In this case, the user will see and interact with only the visible features, but the installation will behave as if all the application resources in the subfeatures are part of the main feature.

Note that marking a subfeature as Required will cause the subfeature to be installed only if its parent feature is installed.

The more features your project has, the more flexibility you have in reinstallation behavior. The REINSTALL property, which should be set during a minor upgrade installation, accepts a list of features to reinstall. When applying a minor upgrade (especially as a patch), you should not use the setting REINSTALL=ALL, but instead explicitly specify the features that you want to reinstall.

A related common practice is to create a top-level "product" feature, as in the figure above.

**Tip 3: Put user-configurable registry data in its own feature.**
When a minor upgrade is applied, all of the registry data in all the features being reinstalled will also be reinstalled; this will occur even if the component containing the registry data is not being updated. This means that any registry settings that have been modified from their original values will revert to their default values. In some cases this is acceptable behavior, but usually you will not want to replace the user's configuration settings with the original factory settings.

If you place user-configurable registry data in its own feature, as described in the previous tip, that feature will not be reinstalled unless it is listed in the value of the REINSTALL property being set during the update.

If you want not to reinstall any registry data, you can also omit the "m" and "u" flags from the REINSTALLMODE value. However, this setting applies to the entire installation, and can have undesirable effects during the application of a patch.

**Tip 4: MSI property values are not automatically saved during the initial installation.**
With a few exceptions, the values of MSI properties that are set during the initial installation will not be available during maintenance mode or an update scenario. If you believe you will need a property's value to be available to a later maintenance or update installation, one common practice is to write the property's value to the registry during the initial installation, and read the data back during the later installation.

To write a property's value to the registry, you can take advantage of the fact that the Value field of the Registry table uses the MSI data type *Formatted*. MSI database fields that use the Formatted data type will expand expressions of the form [PropertyName] into the value of the specified property at run time. For example, to write the account name of the user running the installation into the registry, you can create a value with data "[LogonUser]".

To read back registry data during a later installation, you can populate the AppSearch and RegLocator tables, or use the InstallShield System Search Wizard to populate the tables for you. Of course, you can instead create a custom action script or DLL to read the registry data for you. (You can attach the condition "Not Installed" to an action you want to run only for a first-time installation, and use the condition "Installed" for an action that should run only during a maintenance operation.)

The exceptions mentioned earlier are the values of the MSI properties USERNAME, COMPANYNAME, and ProductID, which are available using the MsiGetUserInfo API function; and the values of ProductVersion and most Programs and Features settings, available with the MsiGetProductInfo API function.

A common requirement is to save the value of the main product installation directory, often stored in the INSTALLDIR property, so that the value is available during a maintenance or update operation. The value of the built-in property ARPINSTALLLOCATION is automatically written to the target system's registry, and is available using the MsiGetProductInfo function. To set ARPINSTALLLOCATION to the value of INSTALLDIR, you can create a set-a-property (Type-51) custom action with source ARPINSTALLLOCATION and target [INSTALLDIR], scheduling it in the Execute sequence after the standard CostFinalize action. If you use InstallShield to create a project, such a custom action (called SetARPINSTALLLOCATION) is automatically included.

### Organizing the Update Project

This section describes techniques involved in authoring common update scenarios. Again, these guidelines are the most relevant for minor upgrades: the uninstall-then-install behavior of a major update reduces your exposure to problems with the design of the original project. If your earlier MSI project has already been deployed, you can often create a major upgrade package to improve the setup design for future updates.

### Deciding Which Type of Update Package to Use

Previously, some of the differences between how minor upgrades and major upgrades are packaged and deployed were described. There are some situations in which a minor upgrade cannot be used, and a major upgrade is required. Some of the cases in which a major upgrade is required are the following:

- If the file name of the MSI database has changed, a major upgrade is required. Therefore, if your organization's build practices include using the product version in the MSI file name (as in *SampleApp-1.2.3.msi*), you will need to use major upgrades to update your product.
- If a component has been removed from an existing feature, or if a component code of an existing component has changed, a major upgrade is required. (Note that this rule applies equally to components in merge modules.)

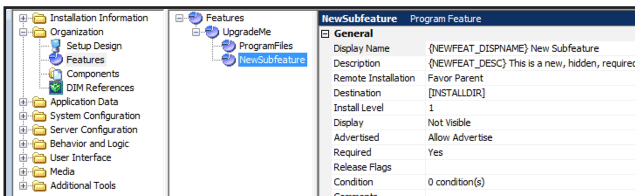
- Similarly, if an existing feature has been moved to become a subfeature of another feature, or if a subfeature has been removed from an existing feature, a major upgrade is required.

Even if you intend to package your update project as a patch, you must usually create a minor or major upgrade package before creating the patch. (An exception is the InstallShield QuickPatch project type.) Recall that an update packaged as a full MSI package can behave as a first-time installation if a particular user does not have an earlier version of your product installed, while a patch package cannot.

If you intend to use patches, it is recommended you create minor upgrades.

**Tip 5: New subfeatures should be marked as "required" and "follow parent".**
A minor upgrade can contain new components in an existing feature. (Very early MSI versions required new components in an update package to be placed in new features, and also required special command-line handling.)

A minor upgrade generally should not take a new top-level feature. However, new subfeatures of existing features are allowed, and should be given the "required" and "follow parent" flags in the Attributes field of the Feature table. In the InstallShield Setup Design view or Features view, set the subfeature's Required property to Yes, and set the Remote Installation property to Favor Parent.



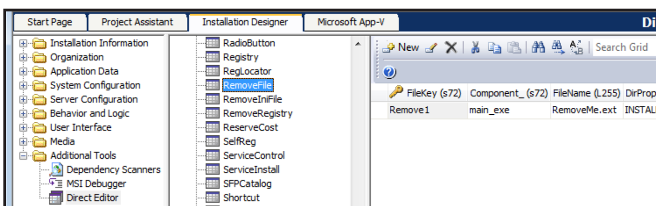

The user interface of a minor upgrade does not usually show the feature tree. Maintenance mode for the updated installation will typically expose the feature tree (in a "Modify" option), and for that reason you might want to mark the new subfeature as hidden. To mark a feature as hidden in the Feature table, enter 0 in the feature's Display field; in InstallShield, set the feature's Display property to Not Visible.

**Tip 6: If your minor upgrade removes data (files, registry settings, and so forth) from a component, populate the corresponding "Remove" data.**
In your minor-upgrade project, removing a file that existed in the earlier product version will not cause the file to be removed when the update is applied. To address this, Windows Installer provides a RemoveFile table, in which you can specify files to remove during installation or uninstallation of the current MSI package.

The fields contained in a RemoveFile record are the following:

- FileKey: a unique, arbitrary primary key for this record (such as "Remove1").
- Component_: reference to a component in the current database; the removal will take place during the installation or uninstallation of this component.
- FileName: the name of the file to remove; you can use wildcard expressions to remove multiple files.
- DirProperty: a property or directory identifier containing the path to the file(s).
- InstallMode: numeric flag indicating when to remove the file(s). Valid values are 1 to remove files when component is installed; 2 to remove files when component is uninstalled; 3 to remove files when component is installed or uninstalled.

In InstallShield, the RemoveFile table is exposed in the Direct Editor view, in the Additional Tools view group. To create a record, click the New button or press Insert, and then populate the fields with the desired data.




Similarly, if you remove registry data from a component in a minor upgrade, you should create a record in the RemoveRegistry table. Records in the RemoveRegistry table describe the registry key and value to remove when the associated component is installed. Unlike the RemoveFile table, the RemoveRegistry table does not accept an option to remove the specified registry data when the associated component is uninstalled. Instead, you can author a registry value with the "uninstall entire key" flag: if your component contains a registry value with a hyphen (-) in the Name field and an empty Value field, the specified registry key and all its contents will be removed when the component is removed.

For other types of data, there is usually either an uninstallation flag available in the MSI table or a corresponding uninstallation table. To remove INI data, for example, there is a RemoveIniFile table; for environment-variable data, there is a corresponding uninstallation flag; and so forth.

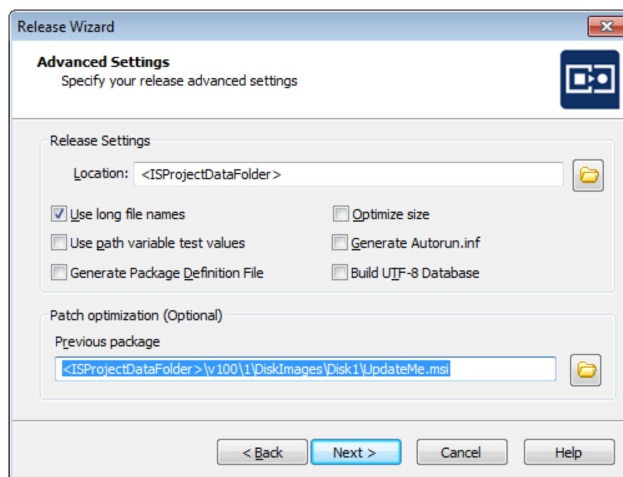
*NOTE: This tip applies only if the component with removed data is private to your product. For components shared with other products, you should change the component code when removing resources. Furthermore, as described above, changing an existing component's component code requires a major upgrade. For more information, see the Windows Installer help library pages "Changing the Component Code" and "What happens if the component rules are broken?"*

InstallShield validates your update packages for appropriate "Remove" data.

**Tip 7: Change the MSI product version for each new release.**
There are some numeric codes that need to be changed in your project for different types of updates. One of these is the MSI product version, stored in the required ProductVersion property. Especially if you intend to package your update as a patch, you will generally want to be able to distinguish an updated version of your package from the original version.

In addition, InstallShield automatically creates a major-upgrade item that prevents an earlier version of your product from being installed over a later version. Changing the ProductVersion each release enables MSI to perform this test.

**Tip 8: When building your update package, use Patch Optimization in your build settings.**
To make the smallest possible patches, file keys in the File table should be identical in the earlier and later MSI databases. The patch-creation process uses the File-table keys to determine if two files are the "same" file. (The actual file names cannot reliably be used, since a package might contain more than one file with the same name, installed under different conditions.)




To use patch optimization in InstallShield, in the last panel of the Release Wizard you can browse for the earlier version of your MSI database.

During the build, InstallShield will ensure the File-table keys are identical for identical files.

**Tip 9: When generating a patch for a compressed installation, use an administrative image.**
The patch-generation process requires uncompressed images of your older and newer installation packages. If your original installation package was built with files compressed, you should generate an uncompressed image by running an *administrative installation*. An administrative installation is not a true installation, in the sense that it does not register any product data on the target system, create shortcuts, write registry data, or register COM servers or file extensions. Instead, an administrative installation simply creates an uncompressed image of an installation.

To run an administrative installation, you can launch the MSI engine executable with the /a switch, as in the following:
msiexec /a ProductName.msi

If your project uses a Setup.exe setup launcher, you can typically also use this command to create an administrative image:
setup /a

The main idea is that you should not create a separate uncompressed build configuration for the sake of patch generation: doing so will compromise the integrity of the File and Media tables between the versions of your installation. Instead, you should always create an administrative image if you need an uncompressed package. InstallShield will automatically create an administrative image for you when you add your installation to a patch configuration in the Patch Design view.

**Tip 10: When generating a patch, both versions should have the same media layout.**
When generating a patch, the earlier and latest versions of your project should both have been built to use compressed files, or both to use uncompressed files. In the case of compressed packages, you should use administrative images to generate the patch, as described in the previous tip.

To understand why this is important, consider a situation where the earlier package was installed using a compressed image. If you create a patch for this installation where the latest version is uncompressed, the patch will transform the cached MSI database on the target system to use references to uncompressed source files. If the patched installation requires the original installation source (for example, because an installed file was accidentally deleted), Windows Installer will make a request for an uncompressed file; and because the original source was compressed, MSI will be unable to find the file to repair it.

Even worse, there are some situations where having mismatched media layouts can cause MSI to delete a good file in an update situation.

For compressed images that span multiple cabinet (CAB) files, then, you should ensure existing files are located in the same CAB file for both the earlier and latest versions. New files can be placed in a new CAB file.

For uncompressed images, files must reside in the same location in the directory structures for the earlier and later versions.

**Tip 11: For a patch, do not set REINSTALL=ALL.**
The REINSTALL property, which should be set during the application of a minor upgrade, can contain a comma-separated list of features to be reinstalled or the special value ALL. However, using the value ALL can cause unwanted prompts for the installation source.

Moreover, the special value ALL reinstalls only those features already installed by an earlier version of the product. During a first-time installation, no features will have been installed, and therefore no features will be installed. For a minor upgrade, if you have a batch file or setup launcher that sets REINSTALL to ALL, you should include a custom action to clear the REINSTALL property for a first-time installation. Another option, handled by InstallShield, is to create an Update.exe setup launcher that tests whether an earlier version of the product has been installed, and sets REINSTALL only when appropriate.

**Tip 12: Updates can contain new dialog boxes and custom actions.**
If you need to handle existing or new data in a special way during an upgrade installation, you can insert new actions and dialog boxes in an update package. The following section describes conditions you can use if you want to run an action only during an upgrade.

## Properties Used in Updates and Patches

In addition to being able to update project files, registry settings, and other data in an update package, you can modify and add dialog boxes and custom actions used in an update package. An update package will run using the sequences defined in the new package.

In some cases, of course, you will want to show certain dialog boxes or perform certain actions only if an update is taking place, and not if the package is a first-time installation. This section describes the various MSI properties used to determine the type of installation taking place. A standard example is the user interface displayed by an installation. By default, a first-time installation displays one sequence of dialog boxes (starting with InstallWelcome); a maintenance mode installation displays another (starting with MaintenanceWelcome); a minor upgrade displays another (SetupResume); and a patch install displays yet another (PatchWelcome).

In the raw MSI database tables (for example, using the Direct Editor view), you can view the “entry point” of a particular series of dialog boxes in the InstallUISequence table. In InstallShield, you can use the Custom Actions and Sequences view, inside the Behavior and Logic view group.

In the InstallUISequence table, only the first dialog box in a series is explicitly listed; subsequent dialog boxes do not appear in the sequence tables, but are instead handled by control events attached to the Back and Next buttons on each dialog box. The Sequences view combines the information represented by the InstallUISequence table and the NewDialog control events attached to Next and Back buttons to display dialog boxes in a tree view.

The key concept is that the same sequence table is used for first-time installations, maintenance mode installations, uninstallation, and so forth: there is no “Uninstall” sequence. The difference when running these different installation modes is that various MSI properties have different values, which indicate what type of installation is appropriate.

In the InstallUISequence table or the Custom Actions and Sequences view, you can review the conditions attached to the entry point of each series of dialog boxes:

- InstallWelcome: Not Installed And (Not PATCH Or IS_MAJOR_UPGRADE).
- MaintenanceWelcome: Installed And Not RESUME And Not Preselected And Not PATCH.
- SetupResume: Installed And (RESUME Or Preselected) And Not PATCH.
- PatchWelcome: PATCH And Not IS_MAJOR_UPGRADE.

The properties involved are:

- *Installed*: set if a product exists on a target system; thus the condition “Not Installed” succeeds for a first-time installation.
- *PATCH*: set if the current installation is packaged as a patch.
- *RESUME*: set if a suspended installation is being resumed, as an installer launched after a reboot caused by the ForceReboot action.
- *Preselected*: set if REINSTALL, ADDLOCAL, or a related property has been set at the command line, indicating a minor upgrade.
- *IS_MAJOR_UPGRADE*: set by InstallShield for a major upgrade (this is not a standard MSI property).

TIP: There are some additional properties you can use to determine if a major upgrade is taking place. Moreover, the Windows Installer API function MsiGetProductInfo (and the equivalent MSI Automation method ProductInfo) can programmatically return information about an installed version of your product, such as its version information, install location, and Programs and Features settings.

## File-Overwrite Rules

A widespread problem with legacy, non-MSI installation programs was that poorly written installers would indiscriminately overwrite existing files on a target system; if an installer replaced a newer version of a file with an older one, existing applications on the target system could fail. To address this problem, Windows Installer enforces strict file-overwrite rules, based on the relationship between the version or modification-date information of the file in the installer and the file on the target system.

In addition to the file-overwrite rules described here, keep in mind that the key file of a component is tested when determining whether to reinstall a component. The simplest file-overwrite rule is that a file with a newer version will replace an existing file with an older version. At installation time, MSI compares the version information in the appropriate File table record to the version of the existing file; if your file has the greater version number, it will be installed. (A special case is that a versioned file will always replace an unversioned file. Moreover, if the versions of the two files are equivalent, MSI performs an additional comparison based on the languages supported by each copy of the file, installing or preserving the file that supports more languages.)

The file-overwrite rules for unversioned files are somewhat more complicated. By default, MSI will not overwrite an unversioned file that has been modified since installation; that is, a file whose creation and modification timestamps are different.

In addition, to prevent unnecessary file-copy operations, Windows Installer will test *file hashes*, if present, for unversioned files. A file hash is a shorthand numeric representation of a file's contents; if two files' hash values are identical, the files' contents are identical. If an unversioned file in an installer has the same hash value as a file on the target system, MSI will not attempt to transfer the file. This behavior is especially useful for patches, where it limits unnecessary prompts for the original installation source.

By default, InstallShield computes file hashes for unversioned files, populating the MsiFileHash table of the MSI database. If you are populating the MsiFileHash table by hand, you will use the MsiGetFileHash API function, or the FileHash method of the MSI Automation interface, to compute the values to enter in your project's MsiFileHash records.

TIP: Another special case involves the relationship between files called *companion files*. Companion files are files that should be installed together: in a companion-file relationship, one file is called the parent, and the other is called the child, and the child is installed whenever the parent is installed. The way you set up the companion-file relationship is to enter, in the Version field of the child's File-table record, the file key of the parent file. In InstallShield, you can set up a companion file relationship by right-clicking the child's file icon in one of the file views, selecting Properties, and entering the primary key of the parent's File record. Note that the child of a companion-file relationship cannot be the key file of its component.

## Changing File-Overwrite Behavior with REINSTALLMODE

The rules described above are the default file-overwrite rules, which apply when the property REINSTALLMODE uses the "o" setting to install over older files on the target system. (Recall that the default value of REINSTALLMODE is "omus".) To change this behavior, you can replace the "o" option with one of the following values:

- *p*: reinstall only if there is no equivalent file on the target system.
- *e*: reinstall if the file is missing or of an older or equal version.
- *d*: reinstall if file is missing or different.
- *a*: reinstall all files, regardless of version.

Keep in mind, however, that the setting for REINSTALLMODE applies to all the features being installed, and cannot be set for an individual feature. In addition, setting REINSTALLMODE to include "a" will likely cause prompts for the original installation source during the application of a patch.

## Summary

In this white paper, you have learned some guidelines for creating initial and updated versions of your installation project, as well as information about the file-overwrite behavior that affects the effectiveness of update installations. You have also read about how InstallShield can help simplify and streamline the installation and update creation process.

**Begin a Free Trial of InstallShield**
You can download a free trial version of InstallShield from the Flexera Software Web site at:
www.flexerasoftware.com/installshield/eval

Want to learn more best practices for building quality installations? Join an InstallShield training class – visit www.flexerasoftware.com/training for available classes.

## How Flexera Software Professional Services Can Help

There are many considerations – technical, operational, and commercial – when designing a reliable yet flexible update/patching strategy. Flexera Software can assist you every step of the way as the world leader in MSI technology with expertise from our background as the developers of InstallShield. Find out more at www.flexerasoftware.com/services/consulting/software-installations.htm.

**Flexera Software LLC**
(Global Headquarters):
+1 800-809-5659

United Kingdom (Europe,
Middle East Headquarters):
+44 870-871-1111
+44 870-873-6300

Australia (Asia,
Pacific Headquarters):
+61 3-9895-2000

Beijing, China:
+86 10-6510-1566

For more office locations visit:
www.flexerasoftware.com

Copyright © 2013 Flexera Software LLC. All other brand and product names mentioned herein may be the trademarks and registered trademarks of their respective owners.

IS_WP_Patching_Oct13