

The Basics of Upgrading Applications Installed using Windows Installer

The following material is a partial excerpt of Chapter 11 of the InstallShield Press book *Practical Windows Installer Solutions for Building InstallShield Setup Applications*.

Written by Windows Installer authority Bob Baker, this book offers an in-depth explanation of the Windows Installer technology and provides expert techniques for creating reliable, customized software installations. Whether you are a Windows Installer novice or an experienced setup developer, you will benefit from the insight on custom actions, installing pure .NET applications, creating upgrades and patches, and installing packages with elevated privileges offered in this expert resource.

To learn more about *Practical Windows Installer Solutions for Building InstallShield Setup Applications* and other InstallShield Press titles, please visit <http://www.installshield.com/ispress>.

The cost of maintaining software applications is usually many times the cost of the original development. Installing software upgrades is a much more common operation than installing the original release of an application. This makes creating effective, reliable upgrades a very important task. Being able to distribute robust upgrades for an application depends a great deal on how the original installation package was structured and distributed.

The Windows Installer technology from Microsoft provides robust functionality that supports the creation of upgrade packages. Windows Installer upgrades can be implemented using either a full installation package or a patch. A patch is special file that contains the difference between a previous version and a new version of an application. A patch just provides an alternate mechanism for implementing the upgrade package than provided by a full installation package. The focus of this excerpt is the use of full installation packages to implement an upgrade but many items that are discussed are also applicable to patch packages.

Important Definitions

Windows Installer defines three types of upgrades, each of which is described below. Each of these upgrade types can be implemented as a full installation package or as a patch.

Small Update. A small update is used to modify a few files for an installed application. The only value that is changed in the MSI package is the package code. The package code (found in the Summary Information Stream sub-view of the General Information view in DevStudio) is the GUID that is stored in the Revision Number property of the summary information stream (SIS). This upgrade type is typically used to deliver bug fixes for a released product. The ProductVersion property is not changed and, because of this, it is difficult to determine if a small update has been applied.

Minor Upgrade. With this upgrade type, the changes made to the installed product are large enough to warrant changing the value of the ProductVersion property. You also need to change the value of the package code GUID stored in the SIS. With this type of upgrade, you could change the minor version value for the product. However, this is not a requirement as long as you change one of the values that compose the ProductVersion property.

Major Upgrade. The major upgrade of a product signifies that the new product offers a significant change in functionality. With this type of upgrade, you need to change the value of the ProductVersion property and the value of the ProductCode property. As with the other upgrade types, you also need to change the value of the package code GUID found in the SIS.

When you change an MSI installation package, you must always change the package code so that it can be used to upgrade a previous application installation. The only time that two packages can have the same package code value is when one of the packages is a copy of the other; in other words, they are the same package. Table 1 summarizes the values that are changed for each upgrade type.

Table 1: Summary of upgrade types and values that are changed for each type.

Upgrade Type	Package Code	ProductVersion	ProductCode
Small	X		
Minor	X	X	
Major	X	X	X

ProductVersion property. This property is one of the five properties that must be given a value in all Windows Installer packages. This property contains the version number of the product and the value of this property has the following format:

```
major.minor.build
```

The first two fields of this property are limited to a maximum value of 255. The last field can have a maximum value of 65,535. When the ProductVersion property is used in a database table column other than one in the Property table, the property can have a fourth field because these columns use the version

data type. However, Windows Installer ignores any fourth field and makes all comparisons using only the first three fields.

ProductCode property: This property is another one of the five properties that must be given a value in all Windows Installer packages. This property is a string GUID that uniquely identifies a particular product release. The ProductCode must be different for different versions of your product and different languages of the same product.

When you change the value of the ProductCode property, you are defining a new product. If this new product is to replace a product that is already installed, you need to perform a major upgrade. There are specific rules defined by Microsoft that guide you in determining when the ProductCode property needs to be changed. These rules are based on how Windows Installer registers information about a product when it is installed.

You can use the rules defined in the following list to determine if you need to change the value of the ProductCode property for your upgrade. You must change the ProductCode property in each of the following cases.

- Your new product needs to coexist on a system with any older versions of the product.
- The name of the MSI package has changed.
- You remove an existing component from an existing feature.
- You make an existing feature a child of another existing feature.
- You remove an existing child feature from its parent feature.
- If you change the component code of an existing component, you have essentially created a new component. When you do this, you need to follow the component creation rules and you need to change the value of the ProductCode property. You need to remove any resources that are no longer used by authoring records in the RemoveFile and RemoveRegistry tables.

UpgradeCode property: This property is not one of the required properties but it is necessary in order to perform major upgrades. This property is a string GUID that uniquely identifies a family of products. For example, you might have an English, German, and French version of your product. Each of the installation packages for these three languages would have a different ProductCode value, but they would all have the same value for the UpgradeCode property.

File overwrite rules: Overwriting files is one of the main operations that occurs during an upgrade. As previously discussed, files are placed on the target system when components are installed. All the files in a component are installed or none of the files in a component is installed. After Windows Installer determines that a component is to be installed or reinstalled, a number of file overwrite rules determine whether a file is replaced. File overwrite rules come into play only when Windows Installer attempts to copy a file to a location where a file with the same name already exists. The rules that Windows Installer uses are global and apply equally to all files in the installation package. There is no mechanism for using different sets of rules for subsets of components.

The Windows Installer file overwrite rules address the following three situations:

1. The file being copied has a version resource. The file with the higher version gets copied over the file with a lower version. A file with a version resource will always copy over a file of the same name that does not have a version resource. A file without a version resource will never be allowed to overwrite a file with a version resource.
2. The file being copied is an unversioned file without a file hash entered into the MsiFileHash table. This type of file is considered user data and will not get overwritten unless the create date and the modified date for the file are the same.
3. The file being copied is an unversioned file that has a file hash entered into the MsiFileHash table. File hashing prevents unnecessary file copying for unversioned files.

Since a file is not installed unless the associated component is installed it is important to review how Windows Installer handles components. The important factors relative to the handling of components are given below:

- A component is not installed or reinstalled unless its associated feature is installed or reinstalled.
- If the condition on a component fails, it is not installed. Unless the transitive bit is set, the condition on a component is not reevaluated during a reinstall.
- If the KeyPath for a component is a versioned file, the component is reinstalled only if the version of the key file is later than the version of the file that has already been installed—unless the default file overwrite rules are modified through the REINSTALLMODE property. In other words, the version of a component is set by the version of its key file. No files in the component are updated unless the key file is updated.
- If the KeyPath of a component is an unversioned file, other factors are used to determine if the files in the component will be updated.
- If the KeyPath for a component is a registry key, registry value, or folder, the files in a component are all updated according to the file overwrite rules that are in effect at the time of the update.

Component Construction

One of the important things that you can do to make sure that any upgrade works as designed is to create your components following the rules set out by Microsoft. A component can be composed of anything that you want to add to the target system. A component can also contain the logic for controlling items already on the target system, such as a Windows service. A component contains resources—which might include files, shortcuts, registry entries—and any other items that can be added to a target system.

Because a component is installed as one unit and removed as one unit, it is important that the resources you place in a component are related to each other. A component is identified by the component code (GUID) that is assigned to it. Two components that have the same component code are considered the same entity even if they contain different resources. Windows Installer handles this situation in two different ways, depending on whether the key path for the component is a versioned file. If the key path is a versioned file, the version of the file defines the version of the component. In this case, a component that is already on the system is not updated with new resources unless the modified component that is

being installed has a key path file with a greater version. If the component has a key path that is not a versioned file or is a folder or registry entry, the modified component modifies the target system with any new resources and uses the file versioning rules to decide whether to replace individual files already on the system.

Two components using the same component code are considered two instances of the same thing and, as such, they must contain exactly the same resources. The corollary to this basic premise is that no two components with different component codes can be used to install the same resource. If this happens for two components, uninstalling one of the components removes the shared resource, disabling the remaining component. This happens because, with two different component codes, each component is assumed to be unique and the registration mechanism has two entries in the registry for the two different component codes. Each component shows that only one product installed it. When one of them is removed from the system, the shared resource is also removed.

You do not need to create a component for every file in your application. Having one component for every file would make the both application and the installation creation process unmanageable. In addition, it would bloat the registry after the application is installed because of how Windows Installer registers components. Microsoft provides a few rules that can help you determine how many components you need to create. These rules cover only a subset of all the files that typically make up an application. You need to determine the component granularity you need to use to create your installation.

The following list provides the basic rules for creating components as defined by Microsoft:

- Never create a component that is already available in a merge module. You should include the merge module in your project instead and make sure that none of the resources in the merge modules are added to any other component that you create in your project. Merge modules are introduced later in this chapter.
- Create a separate component for each .exe, .dll, and .ocx file in your application. In each component, designate these files as the key path for the component. These particular types of files are modified more frequently because they implement the main functionality of an application. It is much easier to distribute a new version of a component if it contains only one of these files than if it contained many.
- Create a separate component for each .hlp and .chm help file. These files must be designated as the key path for the component. The associated .cnt or .chi file should be added to the same component. This allows easier distribution of modified components because one help file is involved in the creation of each component.
- Create a separate component for each file that is the target of a shortcut and make this file the key path of the component. In most cases, this rule is covered by the fact that you need to place every .exe in its own component.
- Any resources, such as registry entries, associated with a particular file should be placed in the same component that contains the file. Because the files in a component can be placed only in a single folder, files that need to be installed to different folders must be placed in different components. It is acceptable, however, to have the files in more than one component installed to the same folder.

These rules provide direction for only a subset of the files that normally make up an application. For the remaining application files, you need to make some decisions before you can place these files into components. The first decision is to assess whether there are any resources that might now ship in the same component, but might be shipped separately in the future. If you can make that determination, you should ship these resources in separate components now.

If you want Windows Installer to be able to check whether a particular file is corrupt, this file needs to be the key path for the component and every file that you feel is important in this respect must be in its own component. This can generate a large number of components and, for a large application, can slow down performance. A large number of components is also harder to manage at build time and bloats the target system's registry at installation. In contrast, you can place the remaining application files in just enough components to match the number of folders into which files need to be copied. This is easier to manage at build time and possibly increases performance when it comes to searching for components. It does not, however, provide the robust self-repair capability that is one of the features of Windows Installer.

Creating components that do not conform to how Windows Installer handles components can result in situations that range from not being able to completely uninstall an application to disabling applications that are already on the system.

The final issue that is important in the construction of components that support upgrades is when you should change the component code and make it a new component. It is acceptable to keep the same component code when revising an existing component if testing shows that the revised component is completely backward compatible with all previous versions of the component. This might be possible for the simplest of components, but it can be argued that true backward compatibility is nonexistent and that any change to a component requires that a new component code be assigned.

If you agree with the supposition that true backward compatibility does not exist, you have to do some additional work when revising a component, in addition to changing its component code. You have to change the name of every resource that is being installed by the new component so they do not conflict with the resources installed by the previous component. This is necessary to avoid overwriting the resources installed by the previous component. These required changes include the following:

- Change the name of the files in the component or change the name of the folder into which the file is to be installed.
- Change the name of the key under which values and data are written.
- Change the name of the shortcut.

For every resource, you need to ensure that both components can exist on the same system without one component overwriting any of the resources installed by the other component.

Preparing for an Upgrade

When you create the installation package for the first release of a product, you should consider possible upgrade scenarios. You can then structure your initial installation so, when it is time to distribute an upgrade, you can minimize the time and effort involved. Regardless of which mechanism—full installation packages or patches—you plan to use to implement upgrades, you need to create your initial and

subsequent installation packages in the same manner. This section provides a number of suggestions that you need to consider relative to creating installation packages that support upgrades.

Note:

InstallShield DevStudio has an Upgrade Validation Wizard that you can access by selecting Validate from the Build menu. The Upgrade Validation Wizard performs a set of tests that validates that your installation will properly upgrade older versions of your product. This wizard is not discussed in this book.

The issues that you need to consider relative to upgrades are described in the following list. Unless otherwise noted, these issues apply to both full installation packages and patch packages as the mode of delivering the upgrade. If you follow these tips, you do not have to make a decision up-front whether to use a full installation package or a patch package to upgrade your product. You need the before and after versions of the product installation to create a patch package. Creating the initial installation packages correctly allows you to use whichever upgrade approach is appropriate.

Design an upgrade-friendly feature tree: Windows Installer performs upgrades on a feature-by-feature basis, whether the upgrade uses a full installation package or a patch package. To provide control over the upgrade operation, it is best to divide an application into logical sets of features, none of which contain a large number of components. The recommended approach is to create one top-level feature and have all other features below this top-level feature.

Maintain a consistent media image (patch packages only): You need to make sure that all versions of your product have the same media layout. This means that if you are compressing your application files in the original version of a product, you need to maintain this approach for all subsequent versions of the product. This is important only if you are distributing your upgrades using a patch package.

Isolate your configurable resources: Normally, during the original application installation, registry entries are made that contain a default set of application configuration values. The end user then configures the application and these values are changed to reflect the user's modifications. When the user installs an upgrade, these user configurations are typically destroyed and everything is set back to the default values. This might be acceptable in some cases, but in most cases it is more practical to maintain a user's customizations. To ensure that the end user's customizations are not destroyed, you can place all registry entries that are customizable in one or more components that are associated with a special feature that is not visible to the end user. When an upgrade is performed, this special feature is not part of the value of the REINSTALL property and thus the feature is not reinstalled during the upgrade.

Author the appropriate tables to remove resources: If you create new versions of your product that do not use resources contained in earlier versions of your product, the upgrade process does not automatically remove the resources that are no longer required. To remove resources as part of an upgrade, you need to author the RemoveFile, RemoveRegistry, and RemoveIniFile tables as appropriate. Because the standard actions that read these tables run before the actions that add resources to the system, the unnecessary resources are removed before new resources are added to the target system.

Whenever possible, use versioned files as the KeyPath for your components: As already discussed, the version of the key file for a component defines the version of the component. The reinstallation of any resource in this type of component depends on whether the version of the key file in the component is greater than the version of the file already installed on the target system. This provides control over what is installed, but it also requires you to adhere to the rules for constructing components. If you want to add additional resources to a component in a later version of your product, you should follow the component construction rules and create a new component with a new component ID.

Always use patch optimization when building your release (patch packages only): The patch creation process uses the keys in the File table of two different packages to determine if two files are the same file. If the file keys are different, the patch is created using the entire file from the later version of the installation package. To create the smallest package possible, it is necessary to maintain a correspondence between the primary keys used in all packages that are part of the upgrade scenario. You can do this in DevStudio in the Advanced Settings panel of the Release Wizard or in the Releases view using the Previous Package attribute. After you do this, the size of any patches you create later are optimized to be as small as possible. Using patch optimization is valuable for any package that implements dynamic file linking with subdirectories because it fixes the component codes to keep them the same from one build to the next.

Change the ProductVersion property for each release: If you do not change the value of this property and you try to apply more than one patch, Windows Installer cannot determine the order of patch application because the version of the product has not changed. If you try to apply a minor upgrade to a product that has a small update using a patch, there is the strong possibility that this upgrade will fail.

If you follow these guidelines, you help ensure that your upgrades will work properly. However, there are two issues that can affect patch creation. First, you cannot create a patch between two packages that have different schemas. Second, you cannot create a patch when your installation packages have more than 32,000 files.

Small Updates and Minor Upgrades Using Full Installation Packages

To provide either a small update or a minor upgrade, you need to run a specific command line when installing the newer package. Typically you would run a command line similar to the following:

```
msiexec /i <path to MSI package> REINSTALLMODE=vomus REINSTALL=ALL
```

The important REINSTALLMODE option used here is the 'v' that tells Windows Installer to cache the new package on the target system. Without this option, the following Windows Installer message box appears when the end user attempts to run the installation (Figure 1).

Note that the above command line also sets another property named REINSTALL. This property identifies the features that are to be installed as part of the upgrade. The value of this property can be a comma-delimited list of feature names or it can be the special value of ALL, which means that all features are to be reinstalled as part of the upgrade. It is important to note that the value of this property pertains only to features that have already been installed by a previous version of the product. If you use the value

of ALL for this property when installing a product for the first time, no installation occurs because no features have already been installed.

Note:

When you use a full installation package to perform an upgrade, you can apply the upgrade only to the image that has been installed to the local machine.

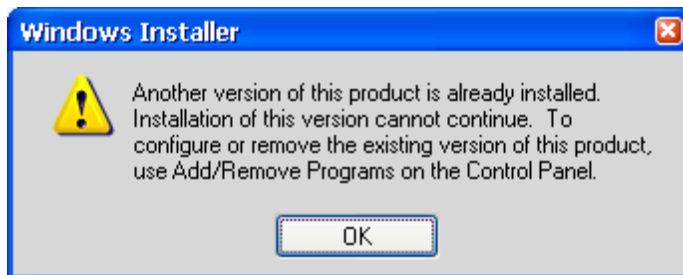


Figure 1: *Installing package with the same ProductCode property as one already installed.*

Your full installation package should be able to run as a fresh installation, as well as an upgrade. To accomplish this you need to be able to detect in advance whether there is a previous version on the target system and set the command line appropriately for the situation encountered. This operation is the responsibility of the bootstrapping application named *setup.exe*. *Setup.exe* can use the *MsiInstallProduct* function to perform a fresh installation or it can use the *MsiReinstallProduct* function to perform an upgrade, depending on whether an earlier version of the product has already been installed.

You can use the REINSTALL property to upgrade only the features that are affected by the change. The REINSTALL property can be set to a comma-delimited list of feature names that need to be reinstalled as part of the upgrade. This type of command line is as follows:

```
msiexec /i <path to MSI package> REINSTALLMODE=vomus  
REINSTALL=<Feature list>
```

This approach is the most efficient because it does not require the complete reinstallation of the product.

Major Upgrades Using Full Installation Packages

When the value of the ProductCode changes, a major upgrade is necessary. A major upgrade is two operations rolled into one installation package. The major upgrade either installs the new version of the product and then silently uninstalls the older version, or silently uninstalls the older version and then installs the newer version. The sequence of these two separate operations depends on how you configure the installation package of the newer version of the product.

The key to generating an installation package that performs a major upgrade is properly authoring the Upgrade table. This table can be authored in DevStudio using the Upgrades view under the Media view

group or you can use the Direct Editor under Additional Tools. The Upgrade table and the standard actions that use this table are discussed in the following two sections.

The sample application referenced here is available on the CD-ROM at the back of the book *Practical Windows Installer Solutions for Building InstallShield Setup Applications*, Bob Baker. Published by InstallShield Press, ISBN 0-9715708-3-3.

THE UPGRADE TABLE

The values that can be placed in the Upgrade table are described in Table 2.

Table 2: *The Upgrade table columns.*

Column Name	Description
UpgradeCode	This GUID is the value of UpgradeCode property that defines the family of products that can be upgraded by the present installation package. The FindRelatedProducts action uses this value to search the target system for any members of the product family that may already be installed.
VersionMin	This value defines the lower bound of the product versions that are to be detected by the FindRelatedProducts action. By default the value entered in this column is not included in the search unless a particular bit-flag is used in the Attributes column. This value can be null as long as the VersionMax column is not null.
VersionMax	This value defines the upper bound of the product versions that are to be detected by the FindRelatedProducts action. By default the value entered in this column is not included in the search unless a particular bit-flag is used in the Attributes column. This value can be null as long as the VersionMin column is not null.
Language	If there are specific product language values that are to be part of the search, these languages are specified in this column. Languages are identified by a comma-delimited list of the language IDs to be included in the search. This column can be null if you want to search for all languages. Also, by setting a specific bit-flag in the Attributes column, you can make any languages in this column exclusive of the search by the FindRelatedProducts action.
Attributes	This column consists of a number of bit-flags that are OR'd together to define how the values in the first four columns are treated. A complete description of the possible bit-flags is provided in Table 3.
Remove	The value in this column is used to set the value of the REMOVE property. The value in this column is a comma-delimited list of feature names that are to be removed. If this column is null, the REMOVE property is set to ALL.

ActionProperty You need to specify the name of a public property in this column. The value of this public property is set by the FindRelatedProducts action with the product codes of the earlier products that are found on the target system. If more than one product code is found, the value of this public property is a semicolon-delimited list of the product codes found. The name of the public property entered in this column needs to be made the value of the special property named SecureCustomProperties.

You can configure how the FindRelatedProducts action treats the values in the Upgrade table by the bit-flags used in the Attributes column. Table 3 describes these bit-flags.

Table 3: *Bit-flags for the Upgrade table Attributes column.*

Value	Description
1	This bit-flag enables the logic in the MigrateFeatureStates action. The MigrateFeatureStates action reads the feature states of the product being upgraded and sets these feature states for the new version of the product being installed. A feature can be in one of the following four states: run locally, run from source, advertised, or absent. This functionality is useful only for a feature tree that is structurally similar to the one being upgraded.
2	Using this bit-flag means that the FindRelatedProducts action searches for existing products that are part of the upgrade family, but these products are not removed during the installation of the new product.
4	Using this bit-flag continues the upgrade product installation even if the older product is not uninstalled successfully.
256	Using this bit-flag makes the product version value in the VersionMin column inclusive. By default, the product version value in the VersionMin column is excluded from the search. If the VersionMin column is null, this bit-flag is ignored.
512	Using this bit-flag makes the product version value in the VersionMax column inclusive. By default, the product version value in the VersionMax column is excluded from the search. If the VersionMax column is null, this bit-flag is ignored.
1024	Using this bit-flag excludes the languages listed in the Languages column in the search for older products. Languages that are not listed in the Languages column are part of the search criteria used by the FindRelatedProducts action.

The standard actions that come into play in a major upgrade and where these actions need to be placed in the sequence tables are discussed in the next section.

THE MAJOR UPGRADE STANDARD ACTIONS

There are three standard actions that are involved during a major upgrade. These actions are described in Table 4.

Table 4: *The major upgrade standard actions.*

Name	Description
FindRelatedProducts	<p>This action begins the major upgrade process. Using the search criteria defined in the Upgrade table, the FindRelatedProducts action searches the target system for any products that match these criteria. If any are found, the value of the ProductCode is made the value of the public property listed in the ActionProperty column of the Upgrade table. This action is placed in both the InstallUISequence and the InstallExecuteSequence tables. However, it does not run in the InstallExecuteSequence table if it has already run in the InstallUISequence table.</p>
MigrateFeatureStates	<p>This action migrates the feature states from the earlier version of the product to the version that is running as the major upgrade. This action also reads the Upgrade table to see if the appropriate bit-flag (1), that requests the migration of the feature states, has been set in the Attributes column. This action is placed in both the InstallUISequence and the InstallExecuteSequence tables. However, it does not run in the InstallExecuteSequence table if it has already run in the InstallUISequence table.</p> <p>This action must run after the FindRelatedProducts action and should also run directly after the CostFinalize action.</p>
RemoveExistingProducts	<p>This action silently removes the earlier versions of the product found by the FindRelatedProducts action. This is accomplished through a proprietary nested install custom action that uses the value of the ProductCode and REMOVE properties. This action is run only during the initial installation of the major upgrade and does not run during any of the possible maintenance operations.</p> <p>The RemoveExistingProducts action is placed only in the InstallExecuteSequence table. However, it can be placed in one of three possible locations in this table. The remainder of this section is devoted to discussing these three possible locations.</p>

The placement of the RemoveExistingProducts action in the InstallExecuteSequence table is guided in large part by whether you have adhered strictly to the rules of component construction. The first possible location for the placement of the RemoveExistingProducts action is shown in Figure 2.

Tables	Action	Condition	Sequence
InstallExecuteSequence	SetARPINSTALLLOCATION	Not Installed	1010
InstallUISequence	SetODBCFolders		1100
IsolatedComponent	MigrateFeatureStates		1200
LaunchCondition	InstallValidate		1400
ListBox	RemoveExistingProducts		1500
ListView	InstallInitialize		1501
LockPermissions	AllocateRegistrySpace	NOT Installed	1550
MIME	ProcessComponents		1600
Media	UnpublishComponents		1700
ModuleComponents	MsiUnpublishAssemblies		1750
ModuleDependency	UnpublishFeatures		1800
ModuleExclusion	StopServices	VersionNT	1900

Figure 2: Placement of the RemoveExistingProducts action before the InstallInitialize action.

When you place the RemoveExistingProducts action before the creation of the execution script, the upgrade removes the earlier product version before attempting to install the major upgrade. This is the location that you want to use if you have not followed the component creation rules carefully and, because of this, it is the default location used by DevStudio. The other locations that you can place this action depend on the component reference-counting mechanism to prevent the removal of components installed by the earlier version of the product that are still needed by the upgraded product.

The problem that can arise with this location is when an error occurs during upgrade installation after the older version has been removed. In this case, the upgrade installation is rolled back and no product is installed on the target system. This location for the RemoveExistingProducts is also inefficient because files that have not changed from the old product version are completely reinstalled during the installation of the new product version.

The placement of the RemoveExistingProducts action as shown in Figure 2 can be set in DevStudio by opening the Upgrades view and selecting the first option in the Major Upgrades Settings group box. This is the option that has the label "Completely uninstall old setup before installing new setup."

The second location for the RemoveExistingProducts action is shown in Figure 3.

Tables	Action	Condition	Sequence
InstallExecuteSequence	StartServices	VersionNT	5900
InstallUISequence	RegisterUser		6000
IsolatedComponent	RegisterProduct		6100
LaunchCondition	PublishComponents		6200
ListBox	MsiPublishAssemblies		6250
ListView	PublishFeatures		6300
LockPermissions	PublishProduct		6400
MIME	ScheduleReboot	ISSCHEDULEREBOOT	6410
Media	InstallExecute		6411
ModuleComponents	RemoveExistingProducts		6599
ModuleDependency	InstallFinalize		6600
ModuleExclusion			

Figure 3: Placement of the RemoveExistingProducts action before the InstallFinalize action.

The location shown in Figure 3 is ideal because it is the most efficient and it provides rollback functionality for both the old product and the new product. As already stated, however, it is critical that all the component construction rules have been followed or the upgraded product might not work properly or not work at all.

In Figure 3, note that just before the RemoveExistingProducts action, a new standard action has been inserted with the name InstallExecute. This action runs the execution script up to its point in the sequence table, but does not terminate the transaction. In the location shown in Figure 3, all changes to the target system are made before running the RemoveExistingProducts action. If the new product uses a number of the files from the earlier product version, these particular files are not copied because of the file overwrite rules. When the RemoveExistingProducts action runs, the reference count for the affected components is decremented by 1.

If an error occurs when installing the upgraded product, it is rolled back before the RemoveExistingProducts action is allowed to run, thus leaving the earlier product version on the target system in working order. If an error occurs while removing the older product, the uninstallation is rolled back, leaving both product versions on the machine in working order.

The placement of the RemoveExistingProducts action as shown in Figure 3 can be set in DevStudio by opening the Upgrades view and selecting the second option in the Major Upgrades Settings group box. This is the option that has the label "Install setup then remove unneeded files." After selecting this option, you also need to select the check box with the label "Rollback all changes if removal of old files fails."

The final location for the RemoveExistingProducts action is shown in Figure 4.

Tables	Action	Condition	Sequence
InstallExecuteSequence	RegisterComPlus		5700
InstallUISequence	InstallServices	VersionNT	5800
IsolatedComponent	StartServices	VersionNT	5900
LaunchCondition	RegisterUser		6000
ListBox	RegisterProduct		6100
ListView	PublishComponents		6200
LockPermissions	MsiPublishAssemblies		6250
MIME	PublishFeatures		6300
Media	PublishProduct		6400
ModuleComponents	ScheduleReboot	ISSCHEDULEREBOOT	6410
ModuleDependency	InstallFinalize		6600
ModuleExclusion	RemoveExistingProducts		6602
ModuleSignature			

Figure 4: Placement of the RemoveExistingProducts action after the InstallFinalize action.

In this location the transaction that installs the product upgrade is complete and can no longer be rolled back. The uninstallation of the older product, however, can be rolled back if there is a problem when it is being uninstalled.

The placement of the RemoveExistingProducts action as shown in Figure 4 can be set in DevStudio by opening the Upgrades view and selecting the second option in the Major Upgrades Settings group box. This is the option that has the label "Install setup then remove unneeded files." After selecting this option you also need to clear the check box with the label "Rollback all changes if removal of old files fails."

Note:

The ALLUSERS property must be the same in both the old and new versions of the installation package. By default, InstallShield DevStudio inserts a custom action named ISSetAllUsers into both the InstallUISequence table and the InstallExecuteSequence table when you create a major upgrade package. When the major upgrade is executed, the ISSetAllUsers custom action finds the value of the ALLUSERS property for the previous version and sets the ALLUSERS property for the upgraded version to this same value.

You have the option of changing this default behavior of DevStudio. You can go to the General tab on the Options dialog and clear the check box at the bottom that has the label "Automatically create ISSetAllUsers action." It is recommended that you retain the default functionality of DevStudio.

SUMMARY OF THE MAJOR UPGRADE PROCESS

The included CD-ROM contains a number of versions of the DevStudio Artist application that you can use to experiment with performing major upgrades. This section walks you through the basic steps in setting up version 2.0 of the DevStudio Artist application to perform a major upgrade of versions 1.0 and 1.1 of the this same application.

1. In the Upgrades view, click the Upgrade Windows Installer Setup node and define the major upgrade settings that you want to use. In this example, the second option is selected with the rollback check box selected. This is shown in Figure 5.

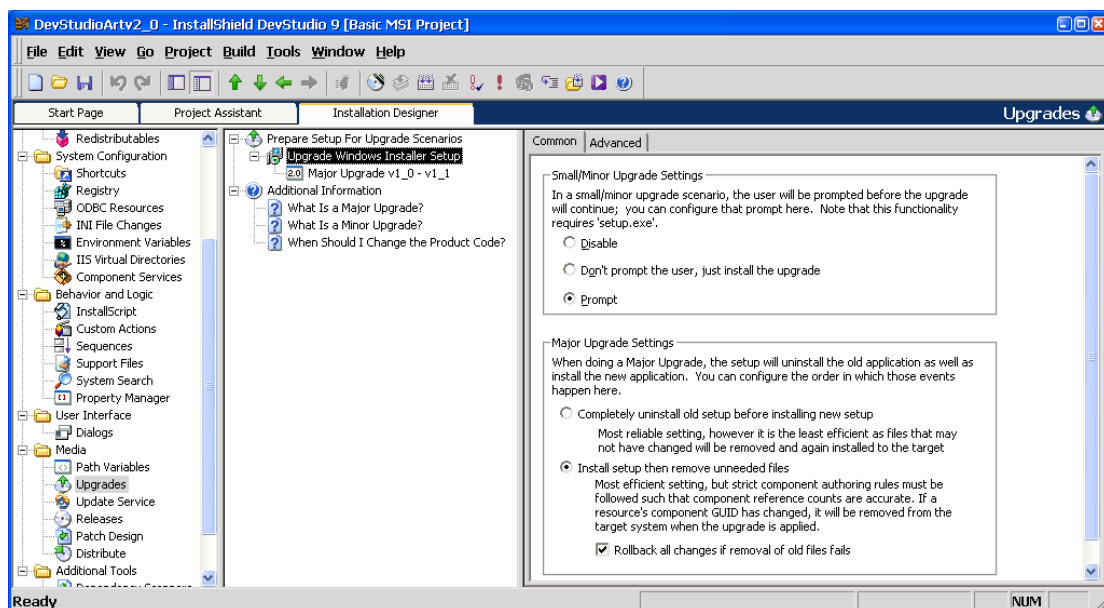


Figure 5: Setting the location of the RemoveExistingProducts action.

- In the Upgrades view, add a major upgrade item under the Upgrade Windows Installer Setup icon in the Prepare Setup for Upgrade Scenarios subview. Set the search criteria to find all products with a version number less than 2.00.0000 as shown in Figure 6.

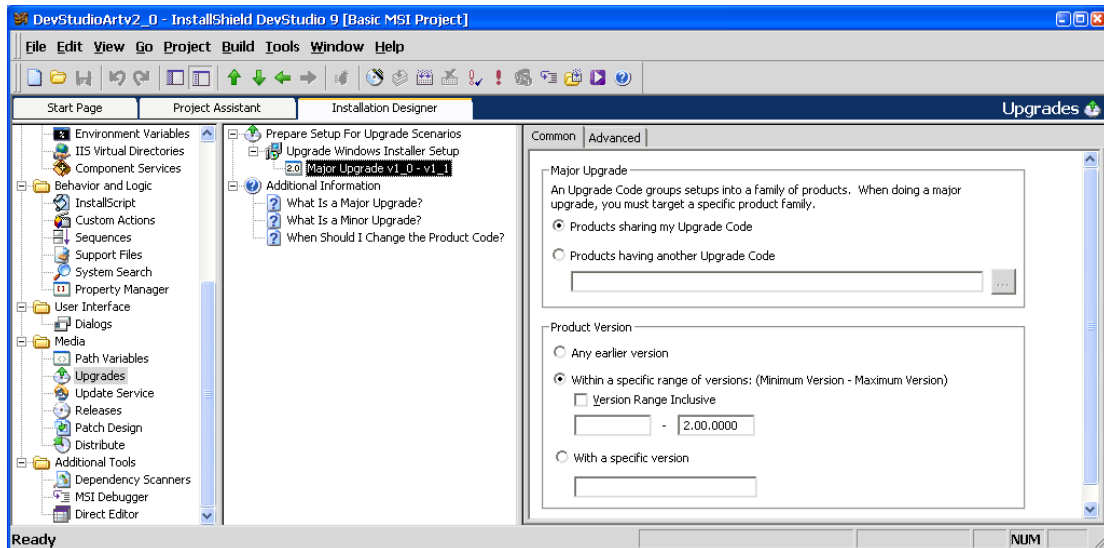


Figure 6: Setting the search criteria for the FindRelatedProducts action.

- Finalize the search criteria on the Advanced tab of the major upgrade item shown in Figure 7. Note that the default name of the ActionProperty property has been changed to make its purpose more obvious.

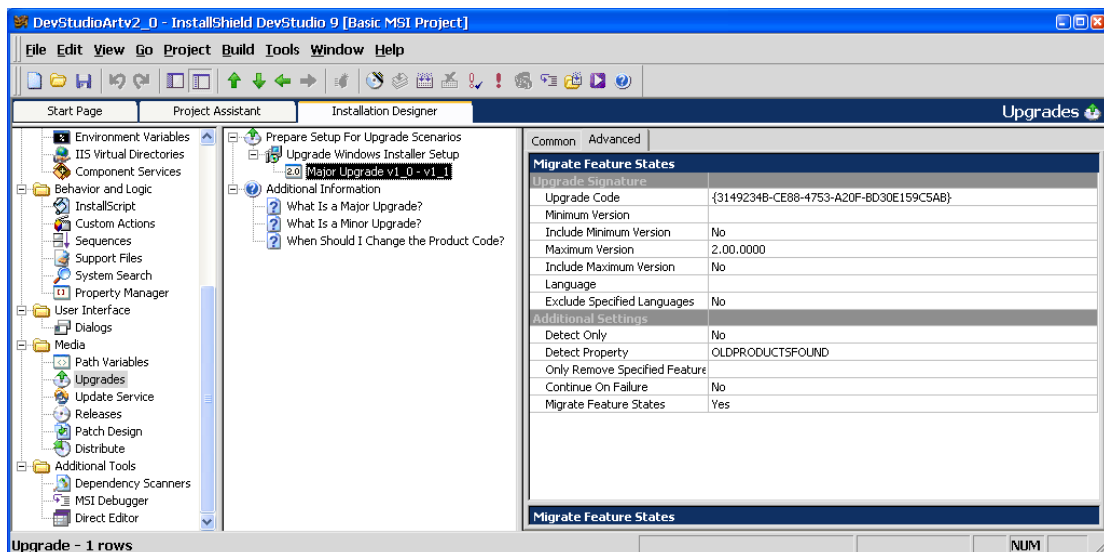


Figure 7: Final search criteria for the FindRelatedProducts action.

After making the settings described above and building the installation package, you will see a row in the Upgrade table that looks like the one shown in Figure 8.

UpgradeCode	VersionMin	VersionMax	Language	Attributes	Remove	ActionProperty
{3149234B-CE88-4753-A20F-BD30E159C5A8}		2.00.0000		1		OLDPRODUCTSFOUND

Figure 8: The Upgrade table in version 2.0 of the DevStudio Artist installation package.

First, install either version 1.0 or 1.0 of DevStudio Artist and then run the installation for version 2.0. When the FindRelatedProducts action is executed, it sets the value of the property OLDPRODUCTSFOUND to the following:

```
{B452E147-4CD8-47F4-BFFC-EB9987304E22}
```

This is the value of the ProductCode property for versions 1.0 and 1.1 of the DevStudio Artist application. When the MigrateFeatureStates action runs, it reads in the Upgrade table the value of 1 in the Attributes column. Because this bit-flag is set, this action sets the states for each feature to be the same as the same feature in the installed version of the product. However, if the Preselected property is set because features have been set on the command line, the MigrateFeatureStates action does not run.

The value of the OLDPRODUCTSFOUND property is sent across to the service process so the RemoveExistingProducts action knows which product to uninstall. To ensure that the value of this property is sent to the service process even in a managed environment, the property OLDPRODUCTSFOUND is set to the value of the SecureCustomProperties property.

Note:

You can identify more than one public property as the value of the SecureCustomProperties property. To do this, separate the names of the public properties with a semicolon.

Preventing Downgrades

You can also use the Upgrade table to prevent a lower version of a product from downgrading a higher version. When you implement the technique described in this section, you can prevent the installation of the older version on any system that has a newer version of the product.

The approach to authoring the Upgrade table is the same as described in the previous section. However, after the FindRelatedProducts has found one or more product code values, you use the name of the ActionProperty property in the Upgrade table as a condition on a Type 19 error custom action. If the ActionProperty has a value, the custom action runs and displays a message that it is not possible to downgrade the product. When the end user clicks the OK button on the message box, the installation terminates.

In version 2.0 of the DevStudio Artist installation package, an additional major upgrade item has been created as shown in Figure 9. Note that in the product version setting, the present version is set as the minimum version that is to be found and that this value is exclusive to the search criteria.

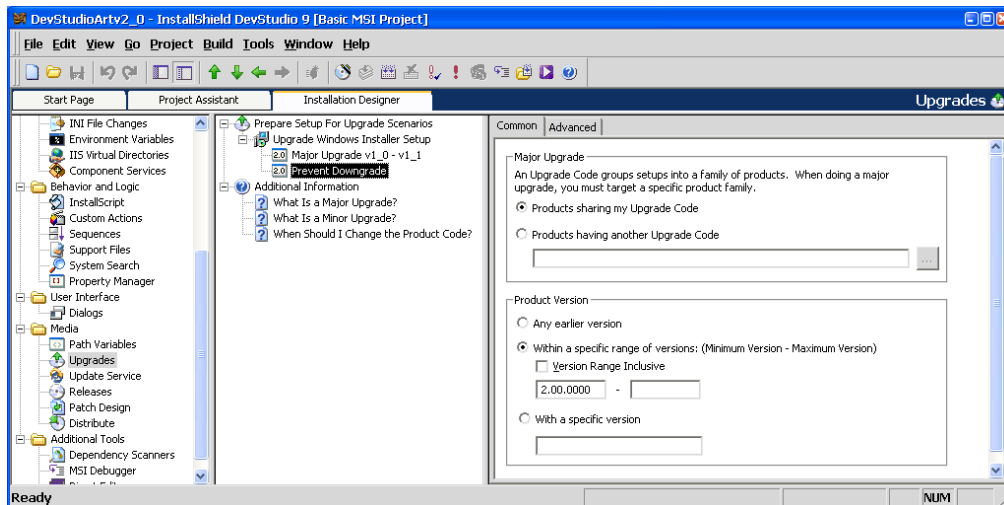


Figure 9: Major upgrade item for preventing a downgrade.

The Advanced settings for this special major upgrade item are shown in Figure 10. You can see that there is a new public property with a name that indicates its purpose. Also note that the Detect Only attribute is set to Yes and the Migrate Feature States attribute is set to No. This is because you only want to identify any later product versions on the target system and do not want to make changes to the system if a later version is installed.

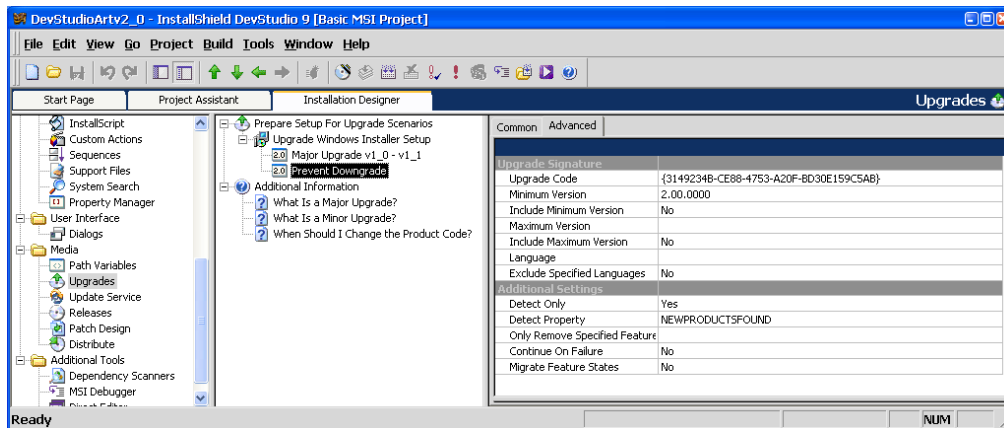


Figure 10: The Advanced settings for the prevention of a downgrade.

Note:

In Figure 9, if you select the "Any Earlier Version" option and you subsequently change the version of your upgrade package, the version number in the Upgrade table is not automatically upgraded. You need to manually change the version number using the Direct Editor.

The final step in preventing a downgrade is to insert a Type 19 custom action in both the InstallUISequence and InstallExecuteSequence tables immediately after the FindRelatedProducts action. This custom action is conditioned using the property NEWPRODUCTSFIND as shown in Figure 11.

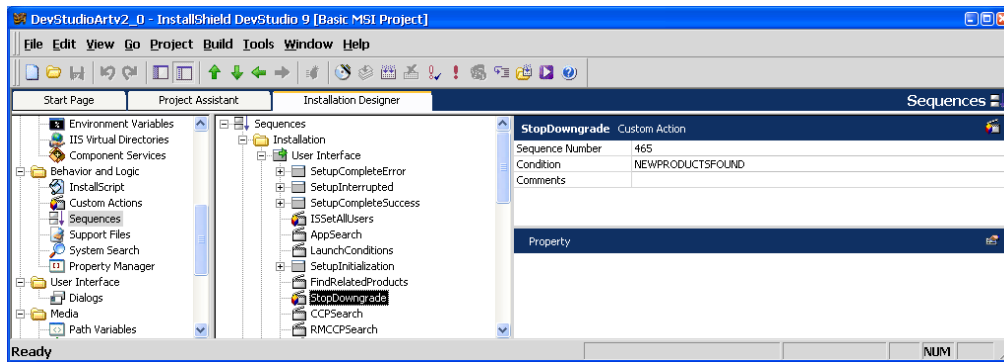


Figure 11: *The Type 19 error custom action placement.*

In DevStudio, a Type 19 custom action can be created only in the Direct Editor. It is the simplest of all the custom actions because it has only one purpose; it displays a message box with an error message and then terminates the installation. This custom action runs only in immediate mode and it takes no processing or scheduling options. The Type column of the CustomAction table has to contain the value 19. The Source column is left blank and the Target column contains a formatted text string. If the formatted text string evaluates to a number, the message displayed is the Message in the Error table associated with the error number. Otherwise, the message displayed is the string entered in the Target field of the CustomAction table.

The above material is a partial excerpt of Chapter 11 of the InstallShield Press book *Practical Windows Installer Solutions for Building InstallShield Setup Applications*.

To learn more about *Practical Windows Installer Solutions for Building InstallShield Setup Applications* and other InstallShield Press titles, please visit <http://www.installshield.com/ispress>.