InstallShield®

WHITE PAPER

MSI Installation Design Issues and Best Practices

by Robert Dickau
Principal Technical Training Writer, Acresso Software



Acresso™ SOFTWARE
Powering the business of software

MSI Installation Design Issues and Best Practices

Introduction

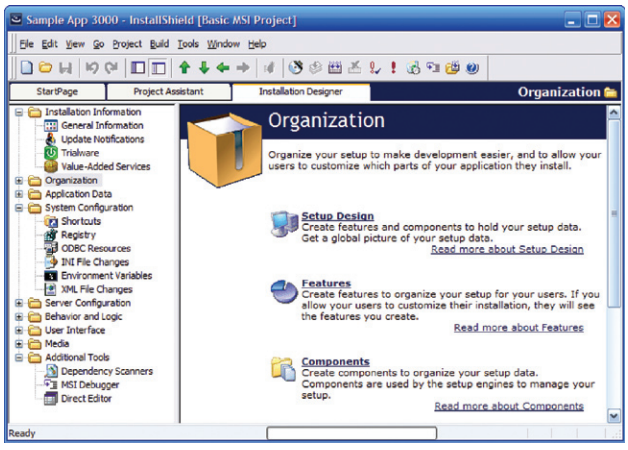
To avoid common errors, there are some techniques and design issues that you should be aware of when working with different parts of the Windows Installer (MSI) technology. This white paper provides an overview of the following Windows Installer concepts related to the design of your installation project, which should help you avoid certain types of problems at build and deployment time:

- Working with features and components
- Working with files
- Working with custom actions
- Working with properties

It also highlights how InstallShield® from Acresso Software assists you in avoiding these problems.

Using the InstallShield Environment

This white paper frequently refers to the InstallShield development environment. It is assumed you are familiar with the general layout of the InstallShield interface, which contains a list of views with which you can modify different portions of your installation project.



For example, the General Information view is where you set general product and project properties; the Setup Design view enables you to edit the features, components, and component data used by your project; the Registry view enables you to modify the registry data installed by your installation program; and the Direct Editor view gives you access to the raw MSI database tables.

It is also assumed you are familiar with some of the wizards available with InstallShield, such as the Release Wizard and Component Wizard.

- The Release Wizard, available under the Build menu and also from the Releases view, lets you describe the properties—media type, compression settings, and so forth—of a release, and then builds the specified release image.
- The Component Wizard, available by right-clicking a feature in the Setup Design view, lets you create special types of components, such as components for COM servers, fonts, and Windows services.

The InstallShield Help Library contains information about using every view and wizard in the InstallShield environment. The InstallShield Help Library is available when you press F1 with any view selected; you can also select Contents from the Help menu to view the help library.

In addition to the graphical environment, InstallShield provides several tools for modifying and building projects from the command line or an external script. For example, to build a project from the command line, batch file, or other automated process, you can use the executable IsCmdBld.exe. The IsCmdBld executable is located in the System subdirectory of the InstallShield distribution directory.

To rebuild a project, you pass IsCmdBld the project file path, the product configuration name, and the release name that you want to rebuild. A sample command appears as follows:

```
iscmdbld -p C:\ProductName.ism -a BuildConfig -r
ReleaseName
```

In addition, InstallShield provides an Automation interface, with which you can modify the contents of a project file without using the graphical environment.

> **Learn More about InstallShield**
> If you wish to learn more about the capabilities of InstallShield, please visit the Acresso Web site at http://www.acresso.com/installshield.

Working with Features and Components

Features represent the end user's view of your installation program, and components represent your view of the installation. Part of your responsibility as the installation designer is to decide how many separately installable pieces of the installation should be presented to the user. Each of these pieces should be a feature or subfeature.

When designing features and components, you should consider the following points:

- All of the file contents of a component must be installed to the same directory. If application files need to be installed in multiple directories, you must create at least one component for each destination.
- A component is the lowest level to which a condition can be attached. There is no method for installing only some of the files in a component, or for installing the files but not registry data for a component. If you have data that need to be installed under different conditions—such as different target operating systems or languages—the data must be separated into different components.
- No resource—file, registry key, shortcut, and so forth—should be placed in more than one component, even across different products and organizations. (Placing the same resource in multiple components breaks Windows Installer reference counting.) Instead, if a resource is required by multiple pieces of the application, you can share entire components among multiple features, or use merge modules to share components among multiple products.
- As described in the following section, for the most effective file transfer, a component should contain at most one executable or DLL—ideally, a versioned file—and this file should be marked as the key file of its component.
- Because component and feature names can occasionally be returned in Windows Installer log files, for the sake of maintainability you should give components and features descriptive internal names. In the case of components, one common practice is to name the component after its key path (usually its key file).
- When there are identical properties for both components and features, the component property is the setting used. For example, both components and features have a Destination property. The feature's Destination property is the one displayed to the end user in the Custom Setup dialog box, while the component's Destination property is the one actually used. In the common special case of the values being the same public property (as in INSTALLDIR), the destination selected in the Custom Setup dialog box is passed along to the component.

Working with Files

In order to avoid or troubleshoot unexpected file installation and uninstallation behavior, it is important to understand the file-overwrite rules that Windows Installer uses, along with the standard uninstallation behavior.

File-Overwrite Rules

When deciding whether to install a component, Windows Installer checks only the key path of the component. If the key path does not need to be replaced, the component will not be installed. For this reason, Windows Installer best practices strongly suggest that the key path of each component be a versioned file, and that each EXE, DLL, and help file be placed in its own component and be marked as the key file.

When testing if one file is "newer" than another—by default, a newer file overwrites an older file—Windows Installer uses the following rules:

- If both files have version resources, the file with the higher version will be installed. Note that if the two files have equal versions, Windows Installer does not check the file dates.
- If one file has a version resource and the other does not, the versioned file will be installed.
- If neither file has a version, if the file on the system has identical creation and modification dates, the file in the installer will overwrite it. Otherwise, the file is assumed to contain user data and settings, and will not be overwritten.

Modifying File-Transfer Behavior

Some of the techniques that exist for modifying the default Windows Installer file-transfer behavior for installation and uninstallation are the following:

- Using the Never Overwrite component setting.
- Changing the REINSTALLMODE property.
- Specifying companion-file relationships.
- Using the Permanent component setting.
- Using the RemoveFile table to delete files created by your application.

You can modify the Never Overwrite setting in the Components view or the Setup Design view of the InstallShield environment. If you set the Never Overwrite component property to Yes, Windows Installer will skip installing the component if its key file is already present on the target system. If the component's key file is not present on the target system, Windows Installer will follow the normal file-overwrite behavior.

You can use the REINSTALLMODE property to modify the default file-overwrite behavior. The REINSTALLMODE property contains a string made of a collection of letters indicating how files should be overwritten. The default value is "omus", where o indicates that older files should be overwritten with new files, m and u indicate that machine and user registry data should be reinstalled, and s indicates that shortcuts should be reinstalled.

To modify file-overwrite behavior, you can replace the o in the REINSTALLMODE value with another letter indicating the desired behavior. Some of the possible values are e, to overwrite a file with an older or equal version, or a, to overwrite existing files without checking their version information.

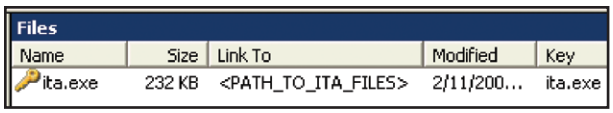
NOTE: The REINSTALLMODE value cannot be limited to a component or feature, but instead applies to all features that will be installed.

To modify the value of REINSTALLMODE, you typically set the value at the command line when deploying the installation. Because REINSTALLMODE is a public property, you can modify its value at the command line as follows:

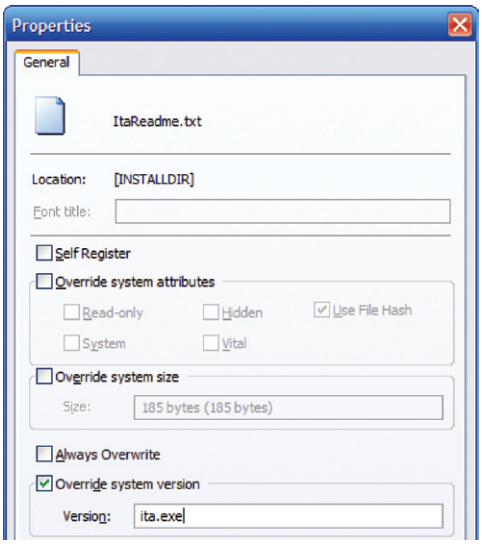
```
msiexec /i product.msi REINSTALLMODE=amus
REINSTALL=ALL
```

(The REINSTALL property contains either a comma-separated list of features to reinstall, or the special value ALL to reinstall every feature already installed on the system.)

Another technique for adjusting file-overwrite behavior is to use companion files, which associate one file's overwrite behavior to another file's behavior. At run time, if the "parent" of the companion file relationship is installed, the "child" will be installed without regard for its version relationship with an existing file. To set up a companion file relationship, for the child's Version data, enter the File-table key of the parent file. By default, the File-table key of a file is displayed in the rightmost (Key) column of the properties in a component's file list.



To enter the parent file's File-table key in the child's version information, you can use the file's Properties panel, available when you right-click the file icon and select Properties. In the Properties panel, select the "Override system version" check box, and then enter the desired Key in the version field.



Equivalently, you can enter the desired key in the Version field of the File-table record using the Direct Editor view.

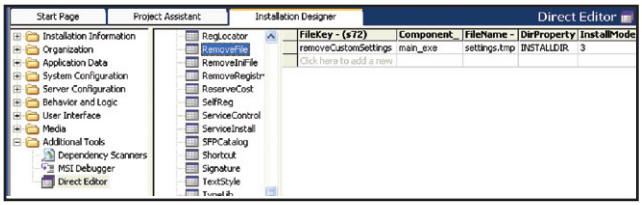
Another aspect of file-transfer behavior is the product's uninstaller. The following techniques enable you to modify your project's uninstall behavior.

If you set the Permanent property to Yes for a component, the component's data will not be removed when the component is uninstalled. By default, when a product is uninstalled, any data installed by the installation will be removed.

By default, a product's uninstallation will remove only files installed by the original installation. Files created by an application after the installation is complete will not be removed automatically. If you want to remove files created by your application, you can add records to the RemoveFile table of your project. The RemoveFile table is where you specify extra files to be removed when a given component is installed or removed.

To add a record to the RemoveFile table, you can use the Direct Editor view. If you select the RemoveFile table icon in the Direct Editor view and press F1, InstallShield opens the Windows Installer Help Library topic that describes the data to enter in a RemoveFile record.

A record in the RemoveFile table appears similar to the following.



There are five fields in a RemoveFile record:

- **FileKey:** An arbitrary primary key ("removeCustomSettings", in the figure above) that uniquely identifies this record.
- **Component_:** The component to associate with this record.
- **FileName:** The name of the file to remove, as in "settings.tmp". You can use wildcard characters in this field to remove multiple files.
- **DirProperty:** The Directory-table property describing the location of the file to delete. In this example, the file is assumed to be in the directory represented by the main installation directory INSTALLDIR. The directory you enter here must be represented by a Directory property, and cannot be a hard-coded path.
- **InstallMode:** A number indicating when to remove the file. The value 1 removes the file when the associated component is installed; 2 removes the file when the component is uninstalled; and 3 removes the file when the component is either installed or uninstalled.

After adding such a record to your project, the installer will delete the file "settings.tmp" from INSTALLDIR when the associated component is installed or removed.

Working with Sequences and Actions

A Windows Installer installation program does not contain an explicit script, but instead the installation behavior is defined by the ordering of standard and custom actions into installation sequences. The order of actions appearing in the sequences determines the order in which your installer performs changes on the target system. An understanding of sequences and actions will help you avoid common problems with the behavior of your installation project.

An installation sequence is divided into two sub-sequences, called the User Interface (UI) sequence and the Execute sequence. The UI sequence displays dialog boxes to the user and queries the target system, and the Execute sequence makes changes to the target system. During a silent or reduced-UI installation, only the Execute sequence is performed; during a test-mode installation, only the UI sequence is performed. (The UI sequence is sometimes called the client process, and the Execute sequence is sometimes called the server process.)

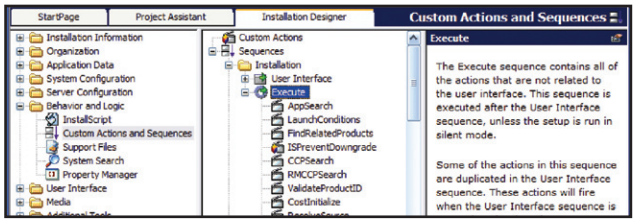
NOTE: Because the User Interface sequence and Execute sequence are two different processes, it is expected that two msiexec.exe processes are visible in the Task Manager.

Standard actions define the built-in Windows Installer functionality. Some of the standard actions Windows Installer uses are:

- InstallFiles
- CreateShortcuts
- WriteRegistryValues
- WriteIniValues

The action names suggest the changes they make to the target system during installation. For details about any specific action, see the Windows Installer Help Library topic describing the action.

To view the order in which actions are performed, you can use the Custom Actions and Sequences view, found in the "Behavior and Logic" view group of the InstallShield environment.

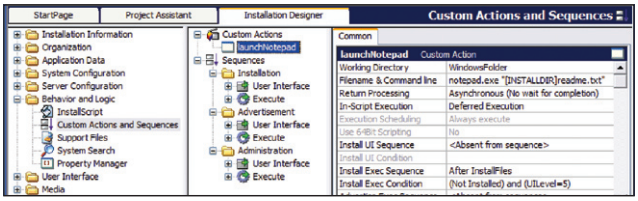


Introduction to Custom Actions

If you need to perform tasks not handled by the standard Windows Installer service actions, you can define many types of custom actions. Custom actions perform tasks such as launching executables, calling DLL functions, or running VBScript or InstallScript code. When working with custom actions, there are two steps to perform:

1. Define the custom action: Specify what it does, whether to check the return value, and so forth.
2. Schedule the action in the installation sequences, relative to other actions in the project.

In the InstallShield environment, you can perform both of these steps in the Custom Actions and Sequences view. To perform the first step, defining the action, right-click the Custom Actions icon, select the desired type of action, and then rename the action. The name you use is the action's internal name, stored in the CustomAction table of your project.



Next, you set the action's properties. (You can also right-click the Custom Actions icon and launch the Custom Action Wizard, in which you enter your action's settings in a series of panels.) For a launch-an-executable custom action, for example, you specify the action's source—whether the executable is already on the target system or bundled within the MSI database—and command-line arguments.

The second step when working with custom actions is to determine where the action runs, relative to other actions in the installation. The following sections describe the various sequencing options available for custom actions.

Sequencing Actions

As described above, a typical installation is controlled by sequences, which are ordered lists of actions the installation performs. The overall flow of an installation is as follows:

- First, Windows Installer starts the installation with the first action in the User Interface sequence with a nonnegative sequence number.
- Actions and dialog boxes in the UI sequence are executed in ascending sequence number until the modeless SetupProgress dialog box is displayed, followed by the ExecuteAction action, which begins the Execute sequence.
- Next, the actions in the Execute sequence are performed in order of ascending sequence number. (As described in the following section, there are two passes through part of the Execute sequence, called immediate mode and deferred mode.)
- When the Execute sequence is complete, control returns to the UI sequence: the SetupProgress dialog box is closed, then the appropriate "setup complete" dialog box (with a negative sequence number) is displayed.

For a custom action, then, you must determine where it should be executed relative to other actions. In the Custom Actions and Sequences view, you can specify where the action should be placed using the Install UI Sequence action setting if you want the action to run during the UI sequence, or the Install Exec Sequence setting if you want the action to run during the Execute sequence. (Alternatively, you can right-click an action icon in the Sequences list and select Insert, and then select the desired custom action. The action you select will be scheduled immediately after the action you right-clicked. You also can right-click a custom action and select Move Up or Move Down to move the action up or down.)

A single action can be executed only once in a given subsequence, but you can schedule an action in both the UI and Execute subsequences, if necessary. Many built-in actions can be found in both sequences, to ensure the action runs at the beginning of either a full-UI installation or a silent installation. You can control the behavior of a custom action that occurs in both the UI and Execute sequences using the Execution Scheduling setting: If you select Always Execute, the action is executed every time it is encountered; the option Execute Only Once executes the action only in the UI sequence for a full-UI installation, and only in the Execute sequence for a silent installation.

Immediate vs. Deferred Execution

Windows Installer makes two passes through the Execute sequence, called immediate mode and deferred mode. Immediate mode is Windows Installer generating its internal script, and deferred mode is Windows Installer performing its internal script.

The internal script is generated during the first pass from the InstallInitialize action through to the InstallFinalize action, and the internal script is carried out during the second pass from InstallInitialize through InstallFinalize. For example, the standard InstallFiles action appears between InstallInitialize and InstallFinalize; during immediate mode, Windows Installer determines which files it will install, and during deferred mode, Windows Installer actually installs the files.

Some guidelines related to determining whether to schedule an action for immediate mode or deferred mode include the following:

- The UI sequence runs only in immediate mode.
- Property values can be set only during immediate mode.
- An action that makes system changes should be scheduled for deferred execution, often in the system context. This is especially important for Windows Vista target systems.
- Similarly, an action that relies on system changes having been made (such as an action that launches an installed executable or document) must be scheduled for deferred execution. For example, a custom action that launches an executable you are installing must be scheduled for deferred execution after InstallFiles.

Deferred execution is valid only between InstallInitialize and InstallFinalize. Windows Installer will report error 2762 if you try to place a deferred custom action in the UI sequence, or in the Execute sequence before InstallInitialize or after InstallFinalize.

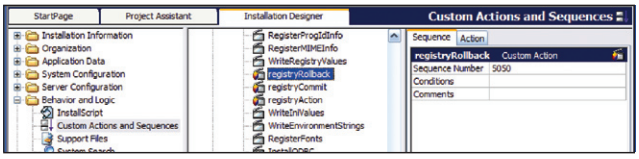
You can perform ICE validation on your MSI database to detect some types of problems with the placement and scheduling of custom actions.

Rollback Execution

As Windows Installer performs system changes using standard actions, it simultaneously creates a rollback script and backs up any data the installer replaces or deletes. If the user cancels the installation during deferred execution, or if the installation fails during deferred execution, Windows Installer consults the rollback script to roll back the system to the state it was in before the installation program ran.

Changes made to the system by custom actions, however, are not logged for uninstallation. Instead, for each deferred action that modifies the target system, you should create a corresponding rollback action to undo the system changes if the installation fails or the user cancels it during deferred execution. (You will typically also need to create a corresponding uninstallation action to undo the system changes when the user uninstalls your application.)

To create a rollback action, you set an action's In-Script Execution setting to Rollback Execution. Note that the rollback action must be placed in your sequences before the action it rolls back. In addition, rollback execution is a type of deferred execution, and therefore must be scheduled between InstallInitialize and InstallFinalize.



One additional action-scheduling option is called commit execution, which executes only when the Execute sequence completes successfully. Rollback actions and commit actions are complementary, in the sense that either the rollback action will run or the commit action will run, but not both. To flag an action as a commit action, set its In-Script Execution setting to Commit Execution.

Working with Properties

To store settings and data that are usable throughout an installation, MSI uses properties. A property is similar to a variable in a programming language. Some properties describe the target operating environment (VersionNT, WindowsBuild, PhysicalMemory), user information (LogonUser, AdminUser, USERNAME, COMPANYNAME), locations of special folders (ProgramFilesFolder, SystemFolder), product information (ProductName, ProductVersion, INSTALLDIR), and installation status and configuration settings (ALLUSERS, ARPINSTALLLOCATION, Installed).

Some properties are stored in the Property table of an MSI database, some are set at the command line by the user or by a setup launcher, and some are initialized by the Windows Installer service when an installation program launches. A property can also be created or set by a custom action, or by a control in a dialog box.

Two classifications of properties are public properties and private properties: Public properties have all-uppercase names (USERNAME, ALLUSERS, MYCUSTOMPROP), and can be set from the command line; private properties have names that contain lowercase letters (Privileged, VersionNT, MyCustomProp), and cannot be changed from the command line.

Perhaps the most common cause of problems related to properties is that only the values of public properties are preserved when installation execution switches from the User Interface sequence to the Execute sequence. (Property names are case-sensitive.) Therefore, if you need the value of a custom property that is defined in the UI sequence (such as one set by a custom dialog box) to be available in the Execute sequence, you should use a public property.

Saving Properties after the Initial Installation

In general, Windows Installer discards the values of properties when an installation completes.

Some exceptions are that many values exposed in the Add or Remove Programs panel are automatically saved by an installation, and are then readable using the MsiGetProductInfo function or ProductInfo method. These properties have names that begin with "ARP", as in ARPHELPLINK, ARPURLINFOABOUT, and ARPHELPTELEPHONE. Similarly, the product version, installation date, and original installation source are stored on the target system to be read back from an external program or custom action using MsiGetProductInfo or ProductInfo.

A common requirement during a maintenance installation or uninstallation is to know the directory in which the application was installed during the initial installation. The value of the predefined property ARPINSTALLLOCATION will be saved to the target system's registry, but a custom action is required to set ARPINSTALLLOCATION to [INSTALLDIR]. InstallShield automatically includes such a custom action: a set-a-property custom action called SetARPINSTALLLOCATION. In a VBScript custom action, for example, the following expression returns the saved value of INSTALLDIR:

```
Installer.ProductInfo(Property("ProductCode"),
"InstallLocation")
```

For other properties, common practice is to save the property value to the registry during the initial installation (by including the form [PROPNAME] in a registry key created by the installation), and then read it back during maintenance. To read the property back, you can set up a system search for the registry data, or use a custom action (with condition Installed, for example).

Conditions

Properties can also be used in Windows Installer conditions. A condition compares the value of a property to another value, or tests whether a condition is defined. The results of a condition can be used to determine, at run time, whether to install a feature or component, or to perform an action or display a dialog box.

Examples of conditions include the following:

- **VersionNT>=500:** Succeeds if the installer is running on Windows 2000 or later.
- **Not Installed:** Succeeds for a first-time installation.
- **PATCH:** Succeeds if a patch is being applied.

When you attach a condition to an action, the action will be performed if the condition succeeds, and will be skipped if the action fails. If there is no condition attached to an action, it will always be performed.

When working with actions, it is important to understand that the same list of actions is called for first-time installations, maintenance operations, and uninstallation. By attaching the appropriate condition to an action, you can specify that the action should run only during a specific type of installation.

To limit actions to a particular type of installation, you can use one of the following conditions:

- **Not Installed:** First-time installation.
- **Installed:** Any maintenance operation.
- **REMOVE="ALL":** Uninstallation (valid after InstallValidate action).

Therefore, if you have a custom action that makes system changes during installation, and you want the effects of the action to be removed during uninstallation, you can give the initial action the condition Not Installed, and give the uninstallation action the condition REMOVE="ALL".

In addition, it is sometimes desirable to tie an action to a feature or component, for example performing the action only when a particular feature is installed. If you want an action to run only when a feature called MainProgram is installed, and it was not already installed, you can use the following condition:

```
&MainProgram=3
```

A condition of the form &FeatureName=n is called a feature action condition, where the condition succeeds only if the feature is changing to a particular state. The number used in the condition indicates the target state of the feature: The value 3 represents "installed locally", and the value 2 represents "absent" or "not installed".

Similarly, you can use component action conditions of the form $ComponentName=n to determine if a component's installation is changing to a specific state. In addition, Windows Installer supports feature-state and component-state conditions of the following forms: !FeatureName=n and

?ComponentName=n. Feature-state and component-state conditions test the state of a feature or component (installed, absent, and so forth) without consideration of whether the state is changing. Feature and component actions are valid only after the CostFinalize action.

For more information, see the Windows Installer Help Library topic "Conditional Statement Syntax".

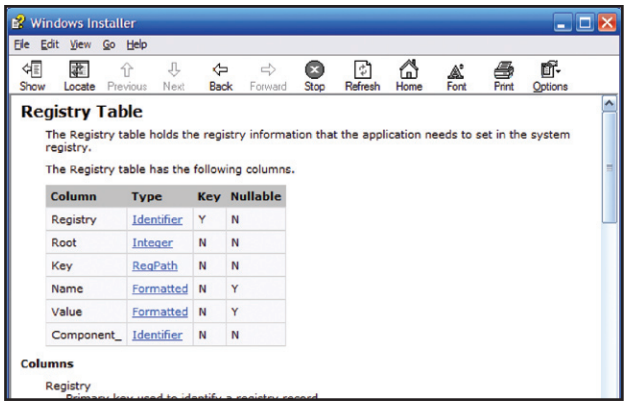
Using the Formatted Data Type

As described previously, in some parts of your project you can use expressions similar to [PropertyName] to expand the value of a property at run time. Taking advantage of the Formatted data type can help you avoid having to write custom actions to write certain types of data to the target system.

The Formatted data type is supported by many fields of MSI database tables. For example, the Name and Value fields in the Registry table support the Formatted data type, which is why you can enter [PROPNAME] in a registry key and have the value written to the registry at installation time. Similarly, the values in the IniFile and Environment tables support Formatted expressions.

Not every table uses the Formatted data type. For example, the name and description of a shortcut do not use the Formatted data type, and therefore one cannot automatically expand [PropertyName] expressions in these shortcut properties. (The target of a shortcut does use the Formatted data type, however.)

To determine if a database field uses the Formatted data type, you can consult the Windows Installer Help Library.



Another common data type is the Identifier type, which accepts only a record identifier. Common uses of the Identifier data type are references to components, features, Directory-table properties, and Property-table properties.

An example is the RemoveFile table, which uses the Identifier data type to store the directory containing the file to be deleted. Because the DirProperty field uses the Identifier data type, INSTALLDIR is a valid value for that field, but [INSTALLDIR] is not.

Properties and Deferred Execution

To get the value of a Windows Installer property in a deferred script custom action, you must populate the CustomActionData property. To populate the CustomActionData property for a deferred action, you perform the following steps:

1. Create an immediate custom action that sets a property (called, for example, ReadPropDeferred) to the value you want to retrieve. In the Custom Actions and Sequences view, for example, create a set-a-property custom action that sets ReadPropDeferred to [INSTALLDIR]. The main idea is that the property name here is the same as the deferred action's name.
2. Next, create a deferred custom action, the name of which is the same as the property you created in the previous step (for this example, ReadPropDeferred). In the deferred action's code, the expression Property("CustomActionData") will return the value you set during the immediate action.

This technique is required only for deferred script custom actions that need to read the value of a property using Session.Property. (This is also required for deferred DLL and InstallScript custom actions that call MsiGetProperty.) Properties will be expanded automatically when used as an argument to a deferred action or in a condition on a deferred action, since these are filled in during immediate mode.

Summary

This white paper discusses some techniques and design issues that you should be aware of when working with different parts of the Windows Installer (MSI) technology, and how InstallShield can help. The overall goal is to help you avoid certain types of problems at build and deployment time by providing an overview of certain Windows Installer concepts.

Begin a Free Evaluation of InstallShield
You can download a free trial version of InstallShield from the Acresso Software Web site at:
www.acresso.com/installshield/eval

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

How Acresso Professional Services Can Help

Acresso has been designing and deploying successful installations for nearly 20 years. With a global team of consultants working in many diverse markets, we can share our experience with you from installation design right through to training. Find out more at www.acresso.com/services/consulting/software-installations.htm.



Powering the business of software

Acresso Software Inc.
1000 E. Woodfield Road,
Suite 400
Schaumburg, IL 60173 USA

Schaumburg (Global Headquarters),
Santa Clara:
+1 800-809-5659

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

Japan (Asia,
Pacific Headquarters):
+81 3-4360-8291

Australia:
+61 2-99-8-22-178

www.acresso.com

© 2009 Acresso Software Inc. and/or InstallShield Co. Inc. All other brand and product names mentioned herein may be the trademarks and registered trademarks of their respective owners.

IS_WP_MSI-Design_Apr09