Chapter
4

The InstallShield Developer Run-Time Architecture

The last chapter provided a detailed description of the Windows Installer technology, which provides the foundation of all InstallShield Developer installations. Chapters 1 and 2 explained that InstallShield Developer can be used to create either a Standard project or a Basic MSI project. The Standard project uses InstallScript on top of the Windows Installer database to create one type of installation program. The Basic MSI project creates an installation that uses the Windows Installer with the ability to use InstallScript in a limited way. In a Basic MSI project, you use InstallScript only to extend the built-in Windows Installer functionality.

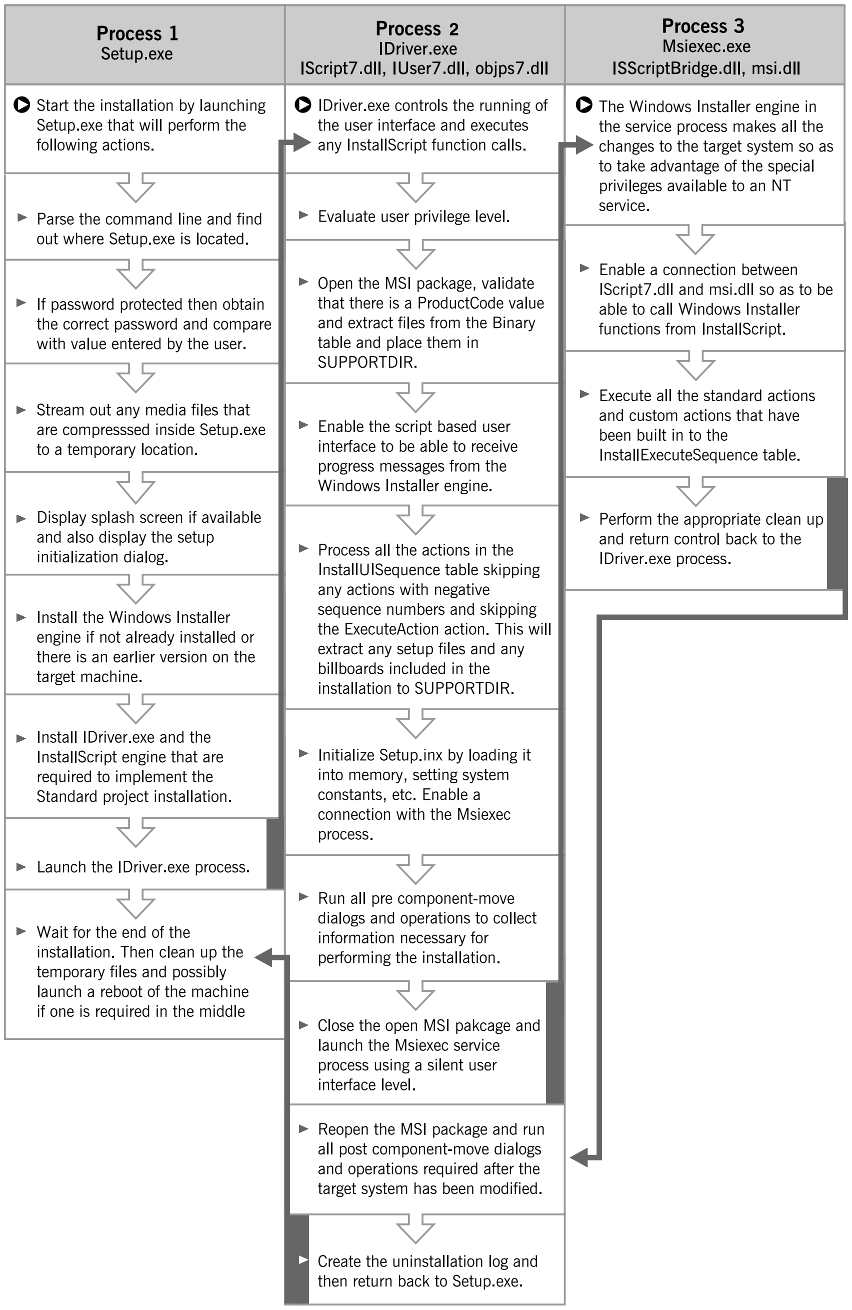
This chapter looks at how InstallShield Developer runs both installation types. We look at the run-time functionality for the typical installation types with emphasis on the differences between the two project types. We begin by discussing the run-time architecture of Standard project installations created with InstallShield Developer. This chapter covers only the basics. You are assumed to be running version 7.03 of InstallShield Developer and that version 2.0 of the Windows Installer engine is installed on your machine. Later versions of InstallShield Developer will probably function differently in some areas but the general concepts provided will be the same. Just like with Chapter 3 you will probably want to reread this chapter after you have worked through some of the examples in this book. The information in this chapter is not critical to learning how to use InstallShield Developer but it does provide background that can be useful when trying to solve problems that arise in the normal course of creating installations.

Fresh Install Run-Time Architecture

An important difference between an installation created using a Standard project and one created using a Basic MSI project is that it is necessary to launch the Standard project installation using setup.exe. For a Basic MSI project, the only time it is necessary to run setup.exe is when all the files are compressed inside. There are a number of other differences, as you will see when we look at how each of the project types implements an installation. We start our discussion with a look at the approach used to implement a fresh install with a Standard project.

Fresh Install Using a Standard Project

When you run a fresh install of a Standard project on Windows NT, Windows 2000, or Windows XP, a minimum of four processes must run in order to implement the installation program. On Windows 9x machines, three processes must run. There are three executables that run in the processes that are used to implement an installation. Setup.exe runs in one of the processes, IDriver.exe runs in another process, and msiexec.exe runs in one or more processes depending on the operating system and other factors.



Process 1
Setup.exe

- Start the installation by launching Setup.exe that will perform the following actions.
- Parse the command line and find out where Setup.exe is located.
- If password protected then obtain the correct password and compare with value entered by the user.
- Stream out any media files that are compresssed inside Setup.exe to a temporary location.
- Display splash screen if available and also display the setup initialization dialog.
- Install the Windows Installer engine if not already installed or there is an earlier version on the target machine.
- Install IDriver.exe and the InstallScript engine that are required to implement the Standard project installation.
- Launch the IDriver.exe process.
- Wait for the end of the installation. Then clean up the temporary files and possibly launch a reboot of the machine if one is required in the middle

Process 2
IDriver.exe
IScript7.dll, IUser7.dll, objps7.dll

- IDriver.exe controls the running of the user interface and executes any InstallScript function calls.
- Evaluate user privilege level.
- Open the MSI package, validate that there is a ProductCode value and extract files from the Binary table and place them in SUPPORTDIR.
- Enable the script based user interface to be able to receive progress messages from the Windows Installer engine.
- Process all the actions in the InstallUISequence table skipping any actions with negative sequence numbers and skipping the ExecuteAction action. This will extract any setup files and any billboards included in the installation to SUPPORTDIR.
- Initialize Setup.inx by loading it into memory, setting system constants, etc. Enable a connection with the Msiexec process.
- Run all pre component-move dialogs and operations to collect information necessary for performing the installation.
- Close the open MSI pakcage and launch the Msiexec service process using a silent user interface level.
- Reopen the MSI package and run all post component-move dialogs and operations required after the target system has been modified.
- Create the uninstallation log and then return back to Setup.exe.

Process 3
Msiexec.exe
ISScriptBridge.dll, msi.dll

- The Windows Installer engine in the service process makes all the changes to the target system so as to take advantage of the special privileges available to an NT service.
- Enable a connection between IScript7.dll and msi.dll so as to be able to call Windows Installer functions from InstallScript.
- Execute all the standard actions and custom actions that have been built in to the InstallExecuteSequence table.
- Perform the appropriate clean up and return control back to the IDriver.exe process.

Figure 4-1: *Standard project fresh install run-time architecture on Windows NT/2000/XP.*

The best way to understand how a fresh install is implemented by a Standard project is to look at a picture and then discuss the elements of the picture in detail. A diagram of three of the four processes that are created when running a Standard project on Windows NT, Windows 2000, or Windows XP is shown in Figure 4-1. The one process that is not shown in Figure 4-1 is only used briefly as part of the initialization of the installation. This fourth process gets generated when the IDriver.exe process opens the database and runs the actions inserted in the InstallUISequence table.

It is assumed here that the Standard project has only an English user interface, that it does not require a reboot in the middle of the installation, and that it is not implementing a Web-based install. We will discuss the additional architectural considerations for a multi-lingual installation later in this chapter. Everything starts with setup.exe, which in turn launches IDriver.exe. IDriver.exe then launches msiexec.exe in a client process and then IDriver.exe launches msiexec.exe in the service process. In this overview, we can safely ignore the msiexec.exe client process because it does not contribute to the actual operations that are being carried out.

The diagram in Figure 4-1 shows that, after execution has moved from process 1 to process 3, it then moves back again from process 3 to process 2 and finally back to process 1 before all operations are complete. For any Standard project installation, setup.exe is the beginning and end of the installation process.

Setup.exe

Setup.exe has many responsibilities in a Standard project installation. Depending on how the installation package is built, setup.exe can have no files streamed into it, a few files streamed into it, or all files streamed into it. The only files that are never streamed into setup.exe are autorun.inf and the SMS Package Definition File (.PDF), when the build is designed to include them. When a build is designed to compress all files into setup.exe, it can be password protected. In this case, setup.exe must compare the password entered by the end user to the correct password before proceeding with the installation.

SETUP.INI

The initialization file Setup.ini provides setup.exe all the information it requires to handle a particular installation. In a Standard installation where there are no files compressed into setup.exe, Setup.ini is included in the media image. If you open the

Setup.ini file for the Standard project that you created in Chapter 2, you will see what is shown in Figure 4-2. In Figure 4-2, the contents of Setup.ini have been annotated.

```
 [Info]
Name=INTL            ;Not used
Version=1.00.000     ;Not used
DiskSpace=8000       ;DiskSpace requirement for temporary files in KB


[Startup]
CmdLine=             ;Can be used to pass command line parameters to
                     ;Setup.exe

SuppressWrongOS=Y    ;Suppresses the display of the warning dialog
                     ;when trying to install version 1.2 of the
                     ;Windows Installer engine on Windows 2000.
                     ;Valid values are Y or N.

ScriptDriven=1       ;Defines whether InstallScript is required
                     ;to run the installation. The following values
                     ;for this keyword are valid
                     ;  0 Basic MSI installation
                     ;  1 Standard project installation
                     ;  2 Basic MSI project using InstallScript
                     ;    custom actions


ScriptVer=7.1.0.179 ;Version of the InstallScript engine required
                     ;for this installation.


Product=Developer Art    ;The name of the product being installed
                         ;for use in the initialization dialog.

PackageName=Developer Art.msi    ;MSI package name for current
                                 ;installation.

MsiVersion=2.0.2600.0    ;Version of the Windows Installer engine
                         ;required for this installation.

EnableLangDlg=N      ;Indicates whether to display the language
                     ;selection dialog to the end user so that the
                     ;language to be used in the user interface can
                     ;be selected. Values are either Y or N.
```

Figure 4-2: *Annotated Setup.ini file for the DeveloperArt_Std project.*

```
DoMaintenance=Y     ;Indicates whether to perform a maintenance
                    ;install or to uninstall the product. The
                    ;possible values are Y or N and apply only to
                    ;Standard projects. This is controlled by the
                    ;Enable Maintenance attribute in the Project
                    ;Properties view of InstallShield Developer.


SuppressReboot=Y    ;Applies to whether the installation of the
                    ;Windows Installer engine version 2.0
                    ;should wait until after the present installation
                    ;is complete. Valid values are Y or N.


[SupportOS]         ;This section identifies the list of operating
Win95=1             ;system sections to follow that provide
Win98=1             ;information regarding the OS properties needed
WinME=1             ;to install the Windows
WinNT4=1            ;Installer version included in this installation.
Win2K=1


[Win95]             ;The attributes of Windows 95 required for
MajorVer=4          ;installing the Windows Installer version
MinorVer=0          ;included in this installation.
MinorVerMax=1
BuildNo=950
PlatformId=1


[Win98]             ;The attributes of Windows 98 required for
MajorVer=4          ;installing the Windows Installer version
MinorVer=10         ;included in this installation.
MinorVerMax=11
BuildNo=1998
PlatformId=1


[WinME]             ;The attributes of Windows ME required for
MajorVer=4          ;installing the Windows Installer version
MinorVer=90         ;included in this installation.
MinorVerMax=91
BuildNo=3000
PlatformId=1
```

Figure 4-2: *Continued.*

```
[WinNT4]              ;The attributes of Windows NT 4.0 required for
MajorVer=4            ;installing the Windows Installer version
MinorVer=0            ;included in this installation. Note that the
BuildNo=1381          ;version of the service pack number, as defined
MinorVerMax=1         ;by the value of the ServicePack keyword, is the
PlatformId=2          ;decimal equivalent of the service pack level in
ServicePack=1536      ;hexadecimal. The value of 1536 is the decimal
                      ;equivalent of 0x600, which means service pack 6.

[Win2K]               ;The attributes of Windows 2000 required for
MajorVer=5            ;installing the version of Windows Installer
MinorVer=0            ;included in this installation.
MinorVerMax=1
BuildNo=2195
PlatformId=2

[Languages]           ;This section identifies the languages available
count=1               ;in this installation that can be selected
default=409           ;by the end user if the language dialog is
enabled.
key0=409


[Developer Art.msi]           ;This section defines the location where
Type=0                        ;the MSI package for this installation
Location=Developer Art.msi    ;is located. The valid values for the
                              ;Type keywords are as follows:
                              ;  0 MSI package on distribution media
                              ;  1 MSI package inside Setup.exe
                              ;  2 MSI in CAB file downloaded from Web
                              ;  3 MSI package installed from Web site
                              ;The location keyword provides the
                              ;URL if this is a Web-based installation,
                              ;otherwise just the name of the package.


[Setup.bmp]      ;This section defines the name and location for the
Type=0           ;file that will be shown as the splash screen at the
                 ;installation’s start. The valid values for
                 ;the Type keywords are as follows:
                 ;  0 The splash screen is on the source media
                 ;  1 The splash screen is inside Setup.exe
                 ;When a splash screen is included, the name is
                 ;specified here. If there is both a language-
                 ;independent and a language-dependent splash screen
                 ;specified, the language-dependent file will be
                 ;displayed.
```

Figure 4-2: *Continued*

```
[instmsiw.exe]          ;This section identifies the location of the
Type=0                  ;Unicode version of the Windows Installer
Location=instmsiw.exe   ;engine. The valid values for the Type
CertKey=MSIEng.isc      ;keyword are as follows:
                        ;  0 Engine located on source media
                        ;  1 Engine located inside setup.exe
                        ;  2 Engine located on Web site
                        ;If version 2.0 of the Windows Installer
                        ;engine is located on the Web site,
                        ; the file identified by the CertKey
                        ;keyword will be streamed into Setup.exe
                        ;so it can be authenticated that
                        ;the Windows Installer engine downloaded
                        ;from the Web comes from Microsoft.

[instmsia.exe]          ;This section identifies the location of the
Type=0                  ;ANSI version of the Windows Installer
Location=instmsia.exe   ;engine. The valid values for the Type
CertKey=MSIEng.isc      ;keyword are as follows:
                        ;  0 Engine located on source media
                        ;  1 Engine located inside setup.exe
                        ;  2 Engine located on Web site
                        ;If version 2.0 of the Windows Installer
                        ;engine is located on the Web site,
                        ;the file identified by the CertKey
                        ;keyword will be streamed into Setup.exe
                        ;so it can be authenticated that
                        ;the Windows Installer engine downloaded
                        ;from the Web comes from Microsoft.

[ISScript.msi]          ;This section identifies the location of the
Type=0                  ;InstallScript engine. The valid values for
Location=isscript.msi   ;the Type keyword are as follows:
                        ;  0 Engine located on source media
                        ;  1 Engine located inside setup.exe
                        ;  2 Engine located on Web site
```

Figure 4-2: *Continued.*

When a password is used to protect the installation by having all files compressed inside setup.exe, the password is added to the Setup.ini file. This entry would look like the following:

```
[KEY]
Password=1953684598
```

The entry that is made in Setup.ini for the password is encrypted into a numerical value. The above example shows how the password "password" is entered into

Setup.ini. When a password-protected setup.exe is launched, it streams out Setup.ini, gets the value of the Password keyword, and then immediately deletes the Setup.ini file. The value held in Setup.ini is decrypted and placed in memory. It is compared against the value that the end user enters in the password dialog box.

The only keyword that you should manually edit in Setup.ini is the `CmdLine` keyword in the `[StartUp]` section. However, when any file is compressed into setup.exe, then Setup.ini is also streamed into setup.exe and is no longer available for post-build modification. The only way to add a value to the `CmdLine` keyword when Setup.ini is to be streamed into setup.exe is to manipulate the Setup.ini template that is used during the build process. The template used by the build process for creating Setup.ini is found in the following location:

```
C:\Program Files\InstallShield\Developer\Support\Setup.ini
```

You can modify this file with the command line that you want to pass to setup.exe. Then, when the setup is built, the value for the `CmdLine` keyword will be included in the Setup.ini file that is streamed into setup.exe.

COMPRESSED MEDIA FILES

The term media files refers to those files that are required for the proper running of the installation but are not part of the files that make up the application being installed. Normally these files consist of the Windows Installer engine, the InstallScript engine installation package, Setup.ini, billboards, splash screen bitmap, etc. Typically these files all reside in the root location of the installation media.

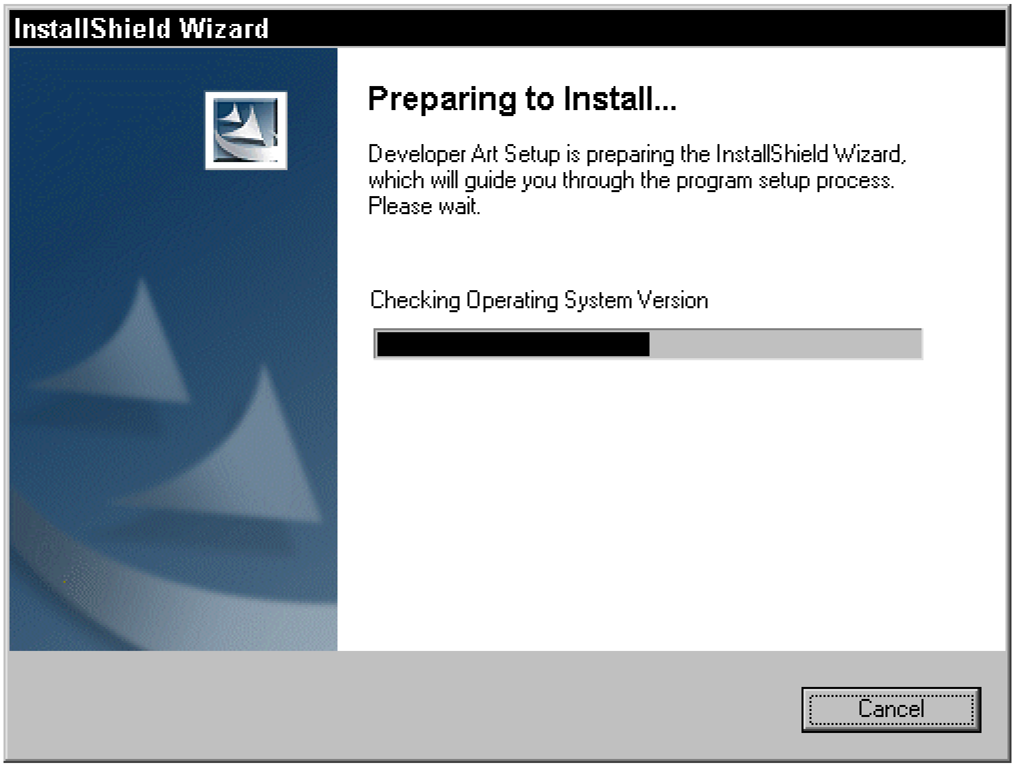
When any of the files that reside on the root of the media are compressed into setup.exe, they have to be streamed out to a temporary directory for use during the installation and they have to be cleaned up after the installation is complete. setup.exe is responsible for implementing both of these operations. These files typically include Setup.ini, the MSI database, the InstallScript engine, the Windows Installer engine, and the splash screen bitmap (if one is included in the installation). The files that are compressed inside setup.exe are streamed out to a location that is uniquely named for each installation that is run. The temporary location used is specified by the TMP or the TEMP environment variable. If neither of these values exists then the temporary location is the Windows folder on Windows NT, Windows 2000, or Windows XP. For Windows 9.x machines the current directory is used if neither these two environment variables are defined. Typically on Windows 2000 this temporary location is as follows, where XXX is a hexadecimal number:

```
%USERPROFILE%\Local Settings\Temp\_isXXX
```

In addition to the files that are streamed out of setup.exe to this temporary location, a file named _ISMSIDEL.INI is created in this directory. This file lists all the files streamed out of setup.exe that have to be deleted after the installation completes. Note that this temporary location is not the one that is given in the SUPPORTDIR system variable.

INITIALIZATION

The main work of setup.exe occurs when the initialization dialog is displayed at the beginning of an installation.


InstallShield Wizard

Preparing to Install...

Developer Art Setup is preparing the InstallShield Wizard, which will guide you through the program setup process. Please wait.

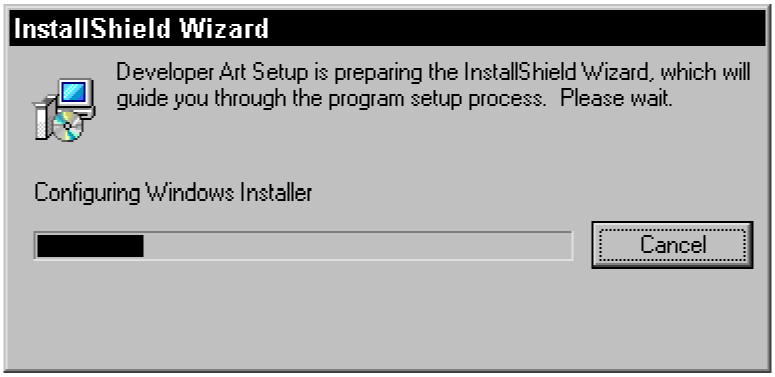
Checking Operating System Version

Figure 4-3: *Large initialization dialog displayed by Setup.exe*

Two different initialization dialogs are included in setup.exe as resources. There is a large dialog that is displayed in the center of the screen when no splash screen is included in the installation (Figure 4-3). The product name displayed in this dialog comes from the value of the `Product` keyword in Setup.ini. This dialog is not displayed if the installation is run silently or if the following entry is made in Setup.ini under the `[Startup]` section..

```
[Startup]
.
.
.
UI=0
```

A small initialization dialog is displayed in the lower-right corner of the screen when a splash bitmap is included in the installation. The splash bitmap is displayed in the center of the screen. This smaller initialization dialog is shown in Figure 4-4.


InstallShield Wizard

Developer Art Setup is preparing the InstallShield Wizard, which will guide you through the program setup process. Please wait.

Configuring Windows Installer

Figure 4-4: *Initialization dialog used when splash screen is displayed.*

The splash screen and the initialization dialog shown in Figure 4-4 are not displayed during a silent installation. The display of the splash screen and small initialization dialog are not affected by the use of the `UI=0` entry in Setup.ini.

The large initialization dialog or the small initialization dialog in conjunction with a splash screen are displayed when an end user runs an installation for a product that is already installed. This triggers the maintenance mode. If maintenance mode is launched from the Add/Remove Programs applet, neither of these initialization dialogs nor the splash screen is displayed. An initialization dialog launched from IDriver.exe is displayed instead.

During the display of the initialization dialog and/or splash screen, three main operations are carried out. These operations are the installation, if necessary, of the Windows Installer engine, the installation of the InstallScript engine, and the launching of IDriver.exe. The installation of the Windows Installer engine starts when the setup program compares the version on the target machine to the version that is included in the installation package. The version included with the installation package is specified by the value of the `MsiVersion` keyword in Setup.ini. If the versions are different or the Windows Installer engine is not already installed on the target machine, the setup program checks the target operating system and compares it to the requirements specified in Setup.ini. If the target system meets the requirements, the Windows Installer engine is installed from the location specified in Setup.ini. If the target system does not meet the requirements for installing the Windows Installer engine, the installation terminates with an error dialog.

Except in one special case, the InstallScript engine is always installed. This engine is installed in such a manner that it cannot be easily uninstalled. The InstallScript engine is installed using the isscript.msi package in silent mode. This chapter takes a closer look at the installation of the InstallScript engine installation package at the end of this chapter. The one exception to always installing the engine is when the InstallScript engine is to be installed from the Web site. In this scenario, the InstallScript engine version on the Web site is checked against the version on the target system and is installed only if it is a later version.

The final operation carried out by setup.exe at the beginning of a Standard project installation is to launch IDriver.exe in a new process. This is accomplished using DCOM because IDriver.exe is a COM server. The IDriver process is created through a call to the `CoCreateInstance` Windows API. This makes the IDriver process a client of the Setup.exe process, thus forcing setup.exe to stay active so that the IDriver.exe process does not prematurely terminate. This second process is where the functions in InstallScript are executed.

When IDriver.exe is launched and the `Install` method is called, setup.exe waits for the installation to complete. When the installation completes, setup.exe has to clean up the temporary directory where all the files were copied that were compressed inside. In addition, setup.exe has to remain active to handle a reboot if one occurs as part of the installation process.

IDriver.exe

All InstallScript code is executed within the IDriver.exe process. It is important to remember this when you use InstallScript to implement program functionality. InstallScript is used in a Standard project to implement the user interface and it can also be used to implement custom actions that are inserted into the InstallExecuteSequence table of the MSI database.

When IDriver.exe starts, it goes through a number of initialization steps that include executing the actions that are in the InstallUISequence table of the MSI database. After initialization is complete, the InstallScript `program...endprogram` function is executed. The `program...endprogram` function is responsible for the user interface, as well as initiating the actions in the InstallExecuteSequence table. Before the IDriver.exe process terminates at the end of an installation, it creates the uninstall log and then passes control back to setup.exe.

INITIALIZATION

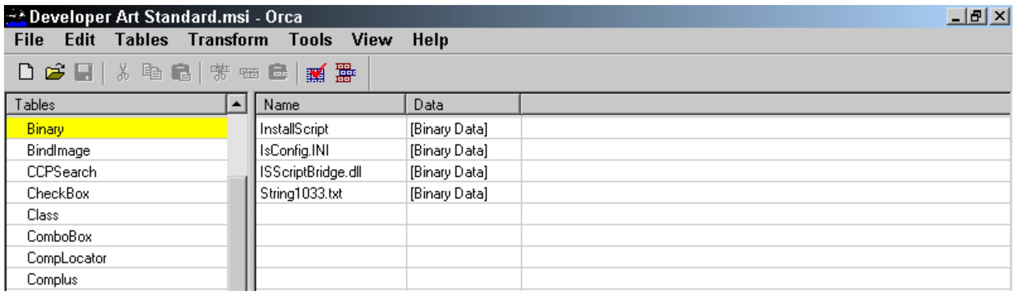
In the initialization process, IDriver.exe first checks if the installation has access to write the uninstall information to the registry. The location to which a Standard project writes the uninstall information is:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\
                        Uninstall\InstallShield Uninstall Information
```

If this registry location is not accessible, the installation is aborted unless elevated privileges have been granted or the installation is being performed for the current user and not for all users of the machine. As soon as this check is completed successfully, IDriver.exe sends initialization progress information to the initialization dialog launched by setup.exe.

The next step in the initialization process is to open the MSI database to verify that a value for the ProductCode property is available and, at the same time, stream out the support files that are contained in the Binary table. The ProductCode is an important entity in the architecture of a Standard project and if it does not exist, the installation cannot continue. If the ProductCode property has no value, the installation aborts.

If you open up the Standard project MSI file that you created in Chapter 2 with Orca and go to the Binary table, you will see what is shown in Figure 4-5.


Developer Art Standard.msi - Orca
File Edit Tables Transform Tools View Help

Tables
Binary
BindImage
CCPSearch
CheckBox
Class
ComboBox
CompLocator
Complus

Name	Data
InstallScript	[Binary Data]
IsConfig.INI	[Binary Data]
ISScriptBridge.dll	[Binary Data]
String1033.txt	[Binary Data]

Figure 4-5: *The Binary table in the Developer Art.msi file*

The identifiers used in the first column of the Binary table are not necessarily the name of the file. The four files in this table are discussed below and referenced by their identifier in the Binary table.

InstallScript: This identifier indicates that this file is the compiled InstallScript. The name of this file when it is streamed out is `setup.inx`.

IsConfig.INI: This identifier indicates an initialization file that enables the Windows Installer engine to call InstallScript custom actions from the msiexec.exe service process that is running the actions in the InstallExecuteSequence table. When streamed out, this file has the same name as the identifier. How this works is discussed in more detail later in this chapter.

ISScriptBridge.dll: This identifier specifies the DLL through which InstallScript custom actions are implemented. This particular file in the Binary table is not streamed out during initialization. The Windows Installer will stream this file out if it is needed.

String1033.txt: This identifier indicates a string table that can be accessed from InstallScript. This string table contains all the pre-defined strings used in a Standard project, as well as all the custom strings that are generated during the authoring process. An example of a custom string is the description of a product feature. This file is streamed out using the same name as the identifier.

Three of the four files in the Binary table are support files and they are streamed out to a temporary folder that is created as part of this operation. The temporary location used is specified by the TMP or the TEMP environment variable. If neither of these values exists then the temporary location is the Windows folder on Windows NT, Windows 2000, or Windows XP. For Windows 9.x machines the current directory is used if neither these two environment variables are defined. Typically on Windows 2000 this temporary location is as follows:

```
%USERPROFILE%\Local Settings\Temp\{ProductCode}
```

This location is the same as described earlier for streaming out media files that are compressed in setup.exe except the name of the folder under the Temp directory is the product code and not a uniquely named folder that changes each time you run the installation. When the system variables are initialized, this location is used to set the value of the SUPPORTDIR system variable that can be accessed from InstallScript.

After the support files have been extracted from the Binary table, the initialization process enables IDriver.exe to receive error messages from the Windows Installer. This is necessary at this time because the next operation is to execute all the actions in the InstallUISequence table. The execution of these actions can then pass back to the initialization dialog any action messages that are produced.

Next in the initialization process, the actions in the InstallUISequence table are executed. As explained in Chapter 3, a sequence table has three columns, the Action, Condition, and Sequence columns. The first step in running the actions in the InstallUISequence table is to perform a SQL query on this table to create a view if all the rows in the table. The SQL query string to do this looks like the following:

```
"SELECT * FROM InstallUISequence ORDER BY Sequence"
```

This SELECT statement obtains all the rows in the InstallUISequence table and the view that is created will have these rows in ascending order of the sequence number assigned to each of the actions in the table. The loop that cycles through each row of the view that is created with the SQL query ignores any action that has a sequence number equal to or less than 0. It also ignores the ExecuteAction action. All other actions in the InstallUISequence table are executed using the Windows Installer function `MsiDoAction` as long as the condition for the action evaluates to TRUE. Chapter 3 provides the basis for understanding this process.

Standard Windows Installer actions and non-InstallScript custom actions can be placed in the InstallUISequence table of a Standard project. You cannot use an InstallScript custom action or place a dialog that has been defined inside the database tables, in the InstallUISequence table. As discussed in Chapter 3, a true Windows

Installer operation executes all actions and dialogs in the InstallUISequence table until it encounters the ExecuteAction action, and then it passes control to the service process. After the service process is completed, the actions following the ExecuteAction action are executed. In a Standard project, all actions before and after the ExecuteAction action are executed as part of the IDriver.exe initialization process before any actions are executed in the InstallExecuteSequence table.

When the actions in the InstallUISequence table are executed, any setup files that have been included in the installation package are extracted. Setup files include billboard bitmaps and dynamic link libraries that are needed during the installation. ISSetupFile is a custom table that holds these setup files, and it is from this table that these files are streamed. The files are extracted when the ISSetupFilesExtract custom action is executed. This custom action is inserted in the InstallUISequence table when setup files are included in the installation. The setup files are extracted to the same folder as the support files discussed earlier. This is the location that is used to define the SUPPORTDIR system variable.

The final step in the initialization process is to load the compiled script into memory and set the values of the system variables. Also in this final initialization step a connection between the IDriver.exe process and the msiexec.exe process is enabled. It is necessary to make this connection so that InstallScript custom actions can be run from the InstallExecuteSequence table.

PROGRAM BLOCK EXECUTION

After all the initialization work is done, IDriver.exe launches the `program` block/function. When you create a Standard project, you do not explicitly create the program block. The `program` block is added to setup.inx at compile time. The `program` block is covered in detail later in this chapter. The program block of code consists of three separate sections, described below.

Pre-component-move operations: Before the installation makes changes to the target system, pre-component-move operations display a user interface and collect information required by the installation. During this phase, the installation should make no attempt to change the target system.

Component-move operations: This section of the `program...endprogram` block launches msiexec.exe in silent mode and initiates the running of the actions in the InstallExecuteSequence table. This is covered in the section entitled Msiexec.exe.

Post-component-move operations: After the target machine has been modified, the installation process typically performs any necessary cleanup of temporary files. One of the operations is the reopening of the MSI package and the rerunning of the file costing actions to reinitialize the Property table. Doing this ensures that property values are available for any InstallScript calls to the `MsiGetProperty` Windows Installer function. Also a dialog must be displayed to show the result of the installation process. There are three primary results that can occur: the installation was completed successfully, the installation was terminated because of an error, or the end user canceled the installation. It is also possible to display a dialog if the installation is causing a reboot but this is not a normal practice. If you want to allow the end user to register the product or some other similar action, this section is where you would incorporate that functionality into the installation.

The `program...endprogram` block is used to call functions, which in turn call the event handlers that you see in the Script Editor and to which you add InstallScript code to perform the actions that are needed in your installation. All changes to the target system need to be implemented in the msiexec.exe process, and it is only in this process that it is possible to roll back the installation. Trying to cancel the installation after control has been returned to the IDriver.exe process should not be permitted.

UNINSTALLATION LOG

For a Standard project installation, all script-related changes to the system are logged to a file called Setup.ilg. If you let the Windows Installer make all the changes to the system, as you should, then the only file that will be logged as having been added to the target system will be the compiled script setup.inx. The compiled script is copied to the system in order to support maintenance operations. The registry entries that are made typically consist of the information required to launch the maintenance session, the location of the log file and the compiled script, and the entry that is made by the InstallScript `SetInstallationInfo` built-in function.

When you let the Windows Installer engine make all the changes to the target system, there are a total of three registry entries that are logged in Setup.ilg. The first of these entries is the information provided under the following key. This information is used by the Add/Remove Programs applet. The key shown here is for the Developer Art installation that was created in Chapter 2.

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\
       Uninstall\InstallShield_{38CE1E93-AD5C-4F9F-800F-607BCB947CE2}
```

The GUID that is the last part of the last key is the value of the ProductCode property. Of course the installation that you created for the Developer Art application will have a different value for the ProductCode property. With a few differences, the value names and values written under this key duplicate the values that the Windows Installer writes in a different location in the registry. One of the values that is written under this key is the location of the Setup.ilg file. The location where the log file is placed on the target system can be controlled through the use of the DISK1TARGET system variable that is available from your script. You can use the Log File Viewer to look inside the log file. The Log file Viewer is launched from the Tools shortcut menu found under Start\Programs\InstallShield.

The second registry entry that is always placed in the log file relates to uninstallation information. For the Developer Art installation, this entry is:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\
                    Uninstall\InstallShield Uninstall Information\
                               {38CE1E93-AD5C-4F9F-800F-607BCB947CE2}
```

The only value used that is written under this registry key is the text string that is displayed in the initialization dialog when a maintenance operation is initiated. The third registry entry that will be created by default is the application information key. This key identifies the name of the company that produced the software, the name of the application that is installed, and the software version. This location in the registry is used to define other keys and values that are necessary for the application to function properly. For the Developer Art application, this entry is:

```
HKEY_LOCAL_MACHINE\SOFTWARE\InstallShield Software Corporation\
                                              Developer Art\1.00.0000
```

Note that there are no values created under this key. To create values in this location, you could use the Registry table in the MSI database and define the values under this key.

There are four system variables that hold values that are written to the log file. You can use three of these system variables in your script to modify the default values that are written to the log file and to the registry. These system variables are discussed in the following list:

UNINST: This system variable contains the command line required to launch IDriver.exe to perform an uninstallation of the product. Even though this system variable is given a default value it is not used. This variable is a hold over from older versions of the InstallShield Professional product and is only provided for the purpose of backward compatibility.

UNINSTALLKEY: This system variable contains the name of the registry key under which all the uninstallation information will be written. For the Developer Art application the default value of this system variable is as follows:

```
InstallShield_{38CE1E93-AD5C-4F9F-800F-607BCB947CE2}
```

UNINSTALL_DISPLAYNAME: This system variable holds the name to be used in the Add/Remove Programs applet. For the Developer Art application the default value of this variable is as follows:

```
Developer Art Standard
```

UNINSTALL_STRING: This system variable contains the command line required to launch IDriver.exe to perform an uninstallation of the product. The default value of this system variable for the Developer Art application is as follows:

```
C:\PROGRA~1\COMMON~1\INSTAL~1\Driver\7\INTEL3~1\IDriver.exe
                     /M{38CE1E93-AD5C-4F9F-800F-607BCB947CE2}
```

You do not want to make any modifications to the UNISTALL system variable because this value is not used. The other three system variables can be modified but you want to make sure that you do not disable the uninstallation functionality for your application.

Msiexec.exe

To make changes to the target system, the Windows Installer engine is launched in silent mode. Chapter 3 explained that the Windows Installer engine runs only the actions in the InstallExecuteSequence table when there is a silent install. On Windows NT, 2000, or XP, the actions in the InstallExecuteSequence table are executed by an

NT service. This permits the administrator to grant elevated installation privileges in a managed environment. In this scenario, a person without administrative privileges can install an application when the system administrator has granted the privilege.

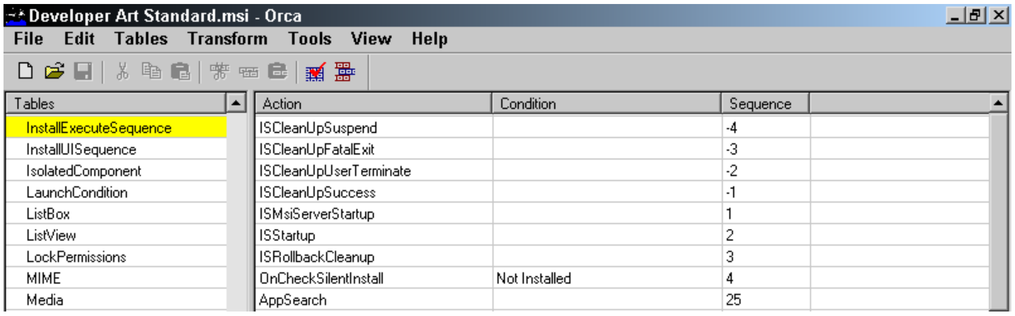
The IDriver.exe process uses the `MsiInstallProduct` function to launch the Windows Installer engine in silent mode. The first two actions with positive sequence numbers in the InstallExecuteSequence table are custom actions that are used to initialize the environment for calling custom actions implemented using InstallScript. A dynamic link library called ISScriptBridge.dll exports the targets of these two custom actions. The first custom action, ISMsiServerStartup, is immediately followed by the ISStartup custom action. These two custom actions work together and need to be the first two actions in the InstallExecuteSequence that have positive sequence numbers.

The ISScriptBridge.dll is streamed into the Binary table at build time and is streamed out of this table by the Windows Installer when it executes the ISMsiServerStartup custom action. This ISMsiServerStartup custom action performs a number of initialization actions. The main purpose of the ISMsiServerStartup custom action is to enable a connection between the msiexec.exe process and the IDriver.exe process. Since all InstallScript code runs in the IDriver.exe process, this connection is necessary whenever any InstallScript custom actions are run. Because of the importance of this custom action and the ISStartup custom action you must not make any changes in the location of these custom actions. They need to stay as the first two custom actions in the InstallExecuteSequence table.

The two-way communication between the msiexec.exe process and the IDriver.exe process is necessary so that you can call Windows Installer functions from within your InstallScript custom actions. Remember that the script engine and the script are running in the IDriver.exe process and the Windows Installer engine, msi.dll, is loaded in the msiexec.exe process. The Windows Installer engine is what exports the Windows Installer functions. Since the functions exported by msi.dll require a handle to the currently running session of the Windows Installer, it is not possible to just load msi.dll into the IDriver.exe process because there would be no valid handle available. Therefore, in order to call Windows Installer functions from a script running in the IDriver.exe process you need to have this two-way communication between these two processes. A more complete discussion of this mechanism is given at the end of this chapter.

When the two-way communication between the IDriver and the Msiexec processes is established, the Windows Installer makes changes to the target system by running the remaining actions, up to the clean-up custom actions. This is the standard approach, discussed in detail in Chapter 3. The final custom action that is executed in the msiexec.exe process is a clean-up custom action. The type of clean up that is performed depends on what happened with the installation. There are four possibilities; the installation was successful, the user terminated the installation before it could finish, there was a Windows Installer error, or the installation was suspended.

In Figure 4-6 you can see that there are four custom actions in the InstallExecuteSequence table that have negative sequence numbers. Part of the functionality of the Windows Installer is to execute one of these actions depending on the outcome of the installation. We have already discussed this mechanism in Chapter 3 as it relates to the showing of the correct dialog at the end of an installation. In the Chapter 3 discussion we were talking about the dialogs that have negative sequence numbers in the InstallUISequence table. The functionality is the same in the InstallExecuteSequence table and in this table this mechanism is used to perform the proper clean up.


Developer Art Standard.msi - Orca
File Edit Tables Transform Tools View Help

Tables	Action	Condition	Sequence
InstallExecuteSequence	ISCleanUpSuspend		-4
InstallUISequence	ISCleanUpFatalExit		-3
IsolatedComponent	ISCleanUpUserTerminate		-2
LaunchCondition	ISCleanUpSuccess		-1
ListBox	ISMsiServerStartup		1
ListView	ISStartup		2
LockPermissions	ISRollbackCleanup		3
MIME	OnCheckSilentInstall	Not Installed	4
Media	AppSearch		25

Figure 4-6: *The clean up custom actions in the InstallExecuteSequence table.*

It is clear from Figure 4-6 which negative sequence number comes into play for each of the possible outcomes of the installation. Since clean up is so important you should not change or remove any of these four custom actions that are associated with the clean up operations.

This completes the overview of the fresh install run-time architecture of a Standard project. We have gone into some detail here to provide a basis for understanding the run-time architecture of other installation modes. Many of the mechanisms already described will apply. The next section discusses how the run-time architecture of a fresh install of a Basic MSI project differs from what we have just covered.

Fresh Install Using a Basic MSI Project

Unlike with a Standard installation project, you can launch a Basic MSI project two different ways. You can use the traditional approach and use setup.exe to launch the installation or you can launch the installation by double-clicking in Windows Explorer on the .msi file. This second approach will only work if the Windows Installer is already installed on the target machine. As you saw in the last section, launching a Standard project installation must begin with running setup.exe.

With a Basic MSI project, there are two scenarios. The first scenario is where there are InstallScript custom actions that have been implemented. The second scenario is where there are no InstallScript custom actions incorporated into the MSI database.

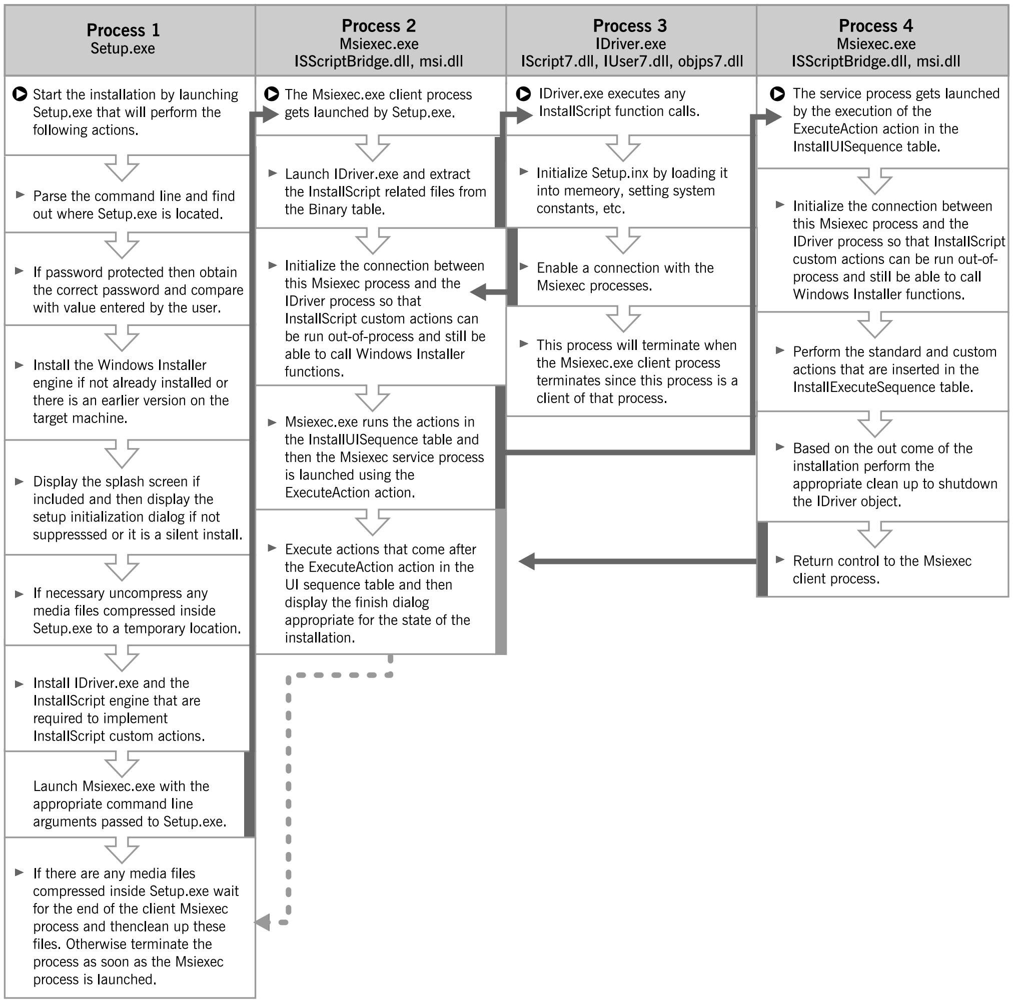
Basic MSI Project With InstallScript Custom Actions

There are four processes involved in this particular scenario. The basic operations that are carried out in these four processes are shown in Figure 4-7.

As with the previous discussion about the run-time architecture for a Standard project, we are only talking about an installation as normally implemented with a full user interface from media with no reboot of the system required during the installation.

SETUP.EXE

There are a few differences in how the Setup.exe process works for a Basic MSI project fresh install compared to how it works for a Standard project fresh install. The entire up-front initialization operations are the same for the two installation types. Any command line options passed to setup.exe with the /v switch are passed on to msiexec.exe. When this operation is complete, the Setup.exe process terminates unless it is required to clean up any media files that were compressed inside it. Media files are compressed and streamed out of setup.exe in the same fashion as described in the section on Standard project installs.



Process 1
Setup.exe

- Start the installation by launching Setup.exe that will perform the following actions.
- Parse the command line and find out where Setup.exe is located.
- If password protected then obtain the correct password and compare with value entered by the user.
- Install the Windows Installer engine if not already installed or there is an earlier version on the target machine.
- Display the splash screen if included and then display the setup initialization dialog if not suppresssed or it is a silent install.
- If necessary uncompress any media files compressed inside Setup.exe to a temporary location.
- Install IDriver.exe and the InstallScript engine that are required to implement InstallScript custom actions.
- Launch Msiexec.exe with the appropriate command line arguments passed to Setup.exe.
- If there are any media files compressed inside Setup.exe wait for the end of the client Msiexec process and thenclean up these files. Otherwise terminate the process as soon as the Msiexec process is launched.

Process 2
Msiexec.exe
ISScriptBridge.dll, msi.dll

- The Msiexec.exe client process gets launched by Setup.exe.
- Launch IDriver.exe and extract the InstallScript related files from the Binary table.
- Initialize the connection between this Msiexec process and the IDriver process so that InstallScript custom actions can be run out-of-process and still be able to call Windows Installer functions.
- Msiexec.exe runs the actions in the InstallUISequence table and then the Msiexec service process is launched using the ExecuteAction action.
- Execute actions that come after the ExecuteAction action in the UI sequence table and then display the finish dialog appropriate for the state of the installation.

Process 3
IDriver.exe
IScript7.dll, IUser7.dll, objps7.dll

- IDriver.exe executes any InstallScript function calls.
- Initialize Setup.inx by loading it into memeory, setting system constants, etc.
- Enable a connection with the Msiexec processes.
- This process will terminate when the Msiexec.exe client process terminates since this process is a client of that process.

Process 4
Msiexec.exe
ISScriptBridge.dll, msi.dll

- The service process gets launched by the execution of the ExecuteAction action in the InstallUISequence table.
- Initialize the connection between this Msiexec process and the IDriver process so that InstallScript custom actions can be run out-of-process and still be able to call Windows Installer functions.
- Perform the standard and custom actions that are inserted in the InstallExecuteSequence table.
- Based on the out come of the installation perform the appropriate clean up to shutdown the IDriver object.
- Return control to the Msiexec client process.

Figure 4-7: *Basic MSI project run-time architecture on Windows NT/2000/XP for a fresh install using InstallScript custom actions.*

MSIEXEC.EXE – CLIENT PROCESS

The operation of msiexec.exe in the client process is the standard Windows Installer approach, as described in Chapter 3. The standard actions, custom actions, and dialogs that are inserted in the InstallUISequence table are executed in ascending order of the positive sequence numbers that were assigned during the installation package's build. The first action in the UI sequence table is the ISMsiServerStartup custom action, with a sequence number of 1. This is a custom action that is implemented in ISScriptBridge.dll and has as its main responsibility the starting of the IDriver.exe process, the extraction of the InstallScript related files from the Binary table, and the initiation of a connection with this process. The script-related files that are streamed in the Binary table are streamed out to a temporary location. This location is set as the value of the SUPPORTDIR InstallScript system variable. This is the same location as described for the Standard project install.

As with a Standard project the purpose of making a two-way connection between the msiexec.exe process and the IDriver.exe process is so that any InstallScript custom actions can be executed in the IDriver.exe process. The connection is reference counted so that IDriver.exe does not terminate prematurely when there are other msiexec.exe processes that still need to run InstallScript custom actions. As already described for a Standard project this two-way connection allows an InstallScript custom action to be able to call Windows Installer functions and access the running database even though they are running in different processes.

Once the two-way connection has been initialized, the Windows Installer processes all the actions and dialogs in the InstallUISequence table, as described in Chapter 3. When msiexec.exe reaches the ExecuteAction action, the running of the InstallExecuteSequence table in the service process begins. When these actions are completed, control returns to the client msiexec.exe process where any final actions coming after the ExecuteAction action are executed and a dialog is displayed indicating that the installation has been completed.

IDRIVER.EXE

The IDriver.exe process for a Basic MSI installation has only a few initialization operations that is has to carry out before it is ready to start executing InstallScript custom actions. The first initialization action is to load the compiled script into memory and to set the values of all the InstallScript system variables. The next and final thing that it does before it is ready to start executing InstallScript custom actions is to enable the connection between itself and the msiexec.exe process.

Once the IDriver.exe process completes the initialization, it waits to receive requests to execute a particular function in the InstallScript code that is loaded into memory. This process services both the client msiexec.exe process and the service msiexec .exe process calls to an InstallScript custom action.

MSIEXEC.EXE – SERVICE PROCESS

When the Windows Installer engine is launched in the service process, it has only one initialization step that has to be performed. The connection with the IDriver.exe process has to be made so as to make Windows Installer functions available to the running script. Since the InstallScript-related files have already been extracted from the Binary table in the msiexec.exe client process, this does not have to be performed here.

With the initialization operations complete, the msiexec.exe service process runs the actions in the InstallExecuteSequence table, as described in Chapter 3. At the end of the execute sequence table, a clean-up custom action runs to shut down the connection to the IDriver.exe process. Once the shutdown is complete, the control of the installation returns to the msiexec.exe client process and any actions that follow the ExecuteAction action are executed. When these are completed, a dialog indicating the results of the installation is displayed. When the end user clicks the Finish button, the installation is over.

Basic MSI Project Without InstallScript Custom Actions

In a Basic MSI project with no InstallScript custom actions, the InstallScript engine is not included as part of the media files. Setup.ini will indicate that there is no script involved with the installation when setup.exe is launched. Setup.exe still performs the same functions described for a Basic MSI project that includes InstallScript custom actions, with the exception of installing the InstallScript engine. The operations that are carried out with this scenario are shown in Figure 4-8. The details of how the Windows Installer implements an installation are discussed in Chapter 3.

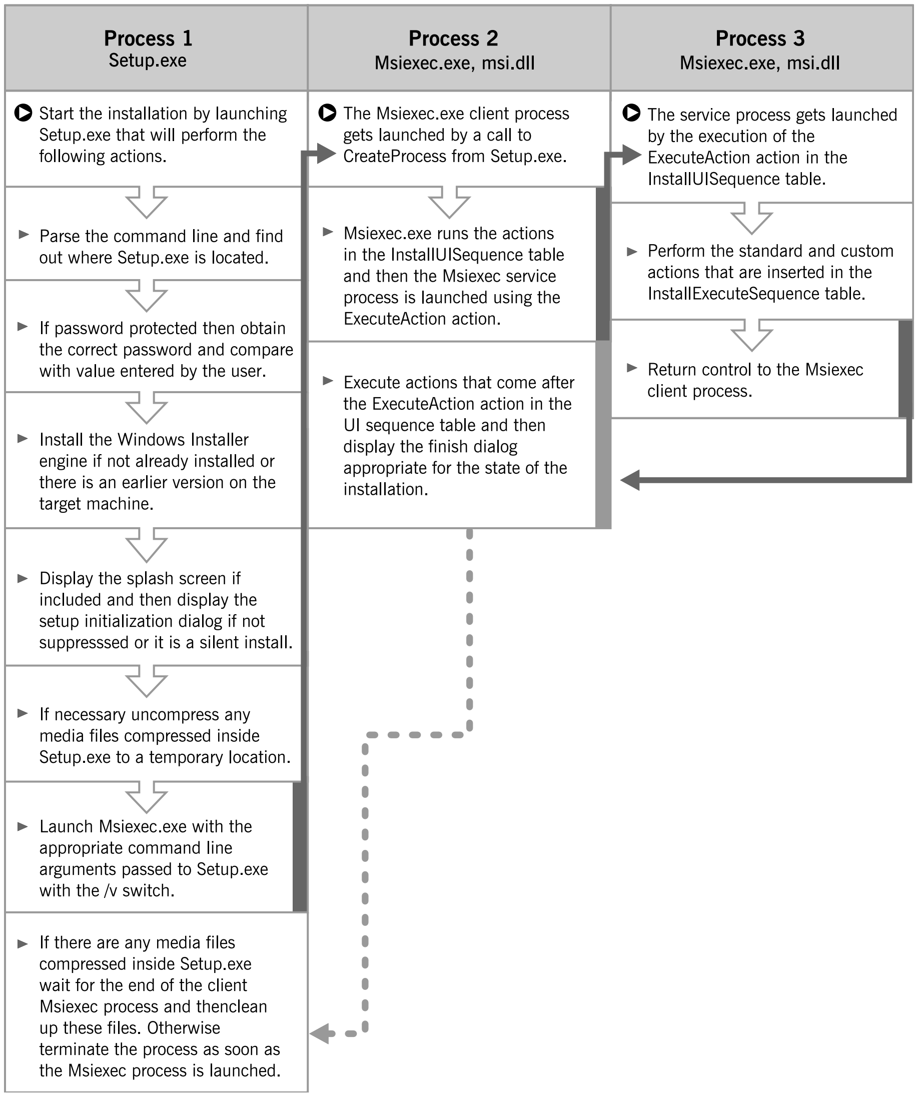

Process 1
Setup.exe
Process 2
Msiexec.exe, msi.dll
Process 3
Msiexec.exe, msi.dll
Start the installation by launching Setup.exe that will perform the following actions.
Parse the command line and find out where Setup.exe is located.
If password protected then obtain the correct password and compare with value entered by the user.
Install the Windows Installer engine if not already installed or there is an earlier version on the target machine.
Display the splash screen if included and then display the setup initialization dialog if not suppresssed or it is a silent install.
If necessary uncompress any media files compressed inside Setup.exe to a temporary location.
Launch Msiexec.exe with the appropriate command line arguments passed to Setup.exe with the /v switch.
If there are any media files compressed inside Setup.exe wait for the end of the client Msiexec process and thenclean up these files. Otherwise terminate the process as soon as the Msiexec process is launched.
The Msiexec.exe client process gets launched by a call to CreateProcess from Setup.exe.
Msiexec.exe runs the actions in the InstallUISequence table and then the Msiexec service process is launched using the ExecuteAction action.
Execute actions that come after the ExecuteAction action in the UI sequence table and then display the finish dialog appropriate for the state of the installation.
The service process gets launched by the execution of the ExecuteAction action in the InstallUISequence table.
Perform the standard and custom actions that are inserted in the InstallExecuteSequence table.
Return control to the Msiexec client process.

Figure 4-8: *Basic MSI project run-time architecture on Windows NT/2000/XP for a fresh install with no InstallScript custom actions.*

Launching a Basic MSI Project From the MSI File

The architecture of a Basic MSI project allows for the launching of the installation by directly launching the .msi file in Windows Explorer or from the command prompt with a command similar to the following:

```
msiexec /i "Developer Art.msi"
```

When a Basic MSI project is launched in this fashion, none of the operations carried out by setup.exe are performed. Such an approach works only if the Windows Installer engine is already present on the target machine. If InstallScript custom actions are used, the InstallScript engine must also be already installed. Without using setup.exe, you cannot password protect your installation. The sole reason that a custom action is used to launch IDriver.exe when InstallScript custom actions are being used is to permit the launching of a Basic MSI project as discussed here.

Maintenance Install Run-Time Architecture

As discussed earlier in the book, when an application is installed for the first time, any further install actions relative to the application come under the heading of maintenance. Normally with either a Standard project or a Basic MSI project, there are three types of possible maintenance operations. These are a Modify operation, a Repair operation, or a Remove operation. These operations are fully discussed in Chapter 1.

The end user can initiate a maintenance installation in two different ways. They can try to run the installation again by running setup.exe or the .msi file if it is a Basic MSI project, or they can use the recommended method of using the Add/Remove Programs applet to launch a maintenance installation.

There is a generic issue for both a Standard and a Basic MSI project when the MSI database and the application files are compressed inside setup.exe. This issue arises when an application is installed where all files are compressed and then the end user tries to perform a maintenance operation that requires additional files to be copied to the target machine. Additional files need to be copied when performing a Repair or a Modify operation that adds a new feature to those that have already been installed.

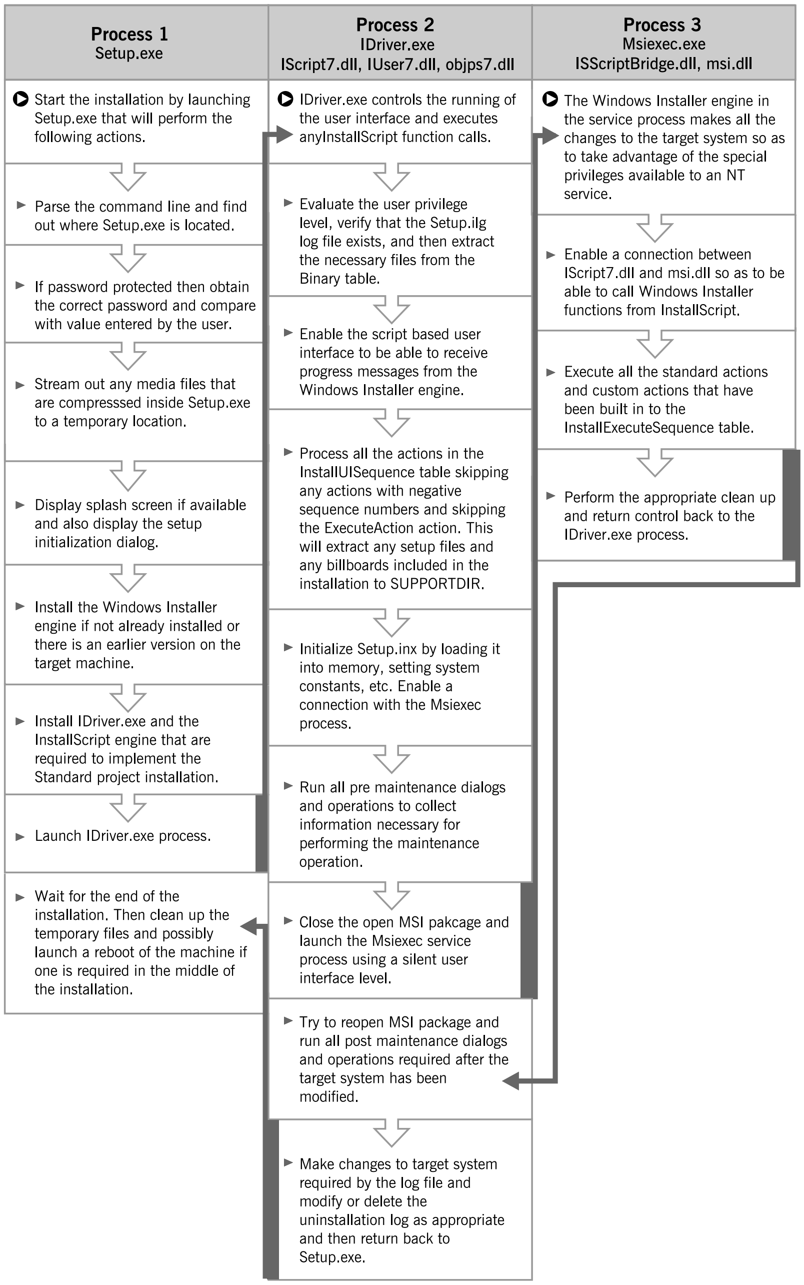
When an end user tries to run this kind of maintenance operation, an error message is displayed to tell the end user that the source is not available. This error occurs because the maintenance operation is looking for the .msi file, which is compressed inside setup.exe, and is not available. The .msi file is also not available for performing maintenance operations, which require the copying of files when the initial installation is performed from a Web site. This potential problem is handled by caching the .msi file on the target system using the /b switch with setup.exe when the end user runs the initial installation. This switch takes as its argument the location where the end user wants the .msi file compressed inside setup.exe to be cached. An example of this command line is as follows:

```
setup /b"C:\InstallCache\Developer Art"
```

When run from the command line like this, the .msi file is copied to the location that is specified after the /b switch and then the installation is run from that location. This will then make the registry entry for the source location be the cached location. With this cached location as the source location, any maintenance operation that needs source files can get them, and the maintenance operation will not fail. This command line switch works for both Standard and Basic MSI projects.

Maintenance Install Using a Standard Project

After an application is installed, the end user can access a maintenance mode by either running setup.exe again or going to the Add/Remove Programs applet. When re-running the installation package launches a maintenance operation, the run-time architecture is as shown in Figure 4-9.



Process 1
Setup.exe

- Start the installation by launching Setup.exe that will perform the following actions.
- Parse the command line and find out where Setup.exe is located.
- If password protected then obtain the correct password and compare with value entered by the user.
- Stream out any media files that are compresssed inside Setup.exe to a temporary location.
- Display splash screen if available and also display the setup initialization dialog.
- Install the Windows Installer engine if not already installed or there is an earlier version on the target machine.
- Install IDriver.exe and the InstallScript engine that are required to implement the Standard project installation.
- Launch IDriver.exe process.
- Wait for the end of the installation. Then clean up the temporary files and possibly launch a reboot of the machine if one is required in the middle of the installation.

Process 2
IDriver.exe
IScript7.dll, IUser7.dll, objps7.dll

- IDriver.exe controls the running of the user interface and executes anyInstallScript function calls.
- Evaluate the user privilege level, verify that the Setup.ilg log file exists, and then extract the necessary files from the Binary table.
- Enable the script based user interface to be able to receive progress messages from the Windows Installer engine.
- Process all the actions in the InstallUISequence table skipping any actions with negative sequence numbers and skipping the ExecuteAction action. This will extract any setup files and any billboards included in the installation to SUPPORTDIR.
- Initialize Setup.inx by loading it into memory, setting system constants, etc. Enable a connection with the Msiexec process.
- Run all pre maintenance dialogs and operations to collect information necessary for performing the maintenance operation.
- Close the open MSI pakcage and launch the Msiexec service process using a silent user interface level.
- Try to reopen MSI package and run all post maintenance dialogs and operations required after the target system has been modified.
- Make changes to target system required by the log file and modify or delete the uninstallation log as appropriate and then return back to Setup.exe.

Process 3
Msiexec.exe
ISScriptBridge.dll, msi.dll

- The Windows Installer engine in the service process makes all the changes to the target system so as to take advantage of the special privileges available to an NT service.
- Enable a connection between IScript7.dll and msi.dll so as to be able to call Windows Installer functions from InstallScript.
- Execute all the standard actions and custom actions that have been built in to the InstallExecuteSequence table.
- Perform the appropriate clean up and return control back to the IDriver.exe process.

Figure 4-9: *Standard project run-time architecture on Windows NT/2000/XP for a maintenance installation initiated from setup.exe.*

There are three processes that run in this type of maintenance scenario. Setup.exe runs just as if this were a fresh install. It displays a password dialog if necessary, displays a splash screen if one is included, and installs the InstallScript engine. There is no difference in the function of setup.exe from what was shown for a fresh install of a Standard project (Figure 4-1).

Setup.exe instantiates the IDriver.exe process and this gets the IDriver.exe process performing all the required initialization operations. Unlike with setup.exe, there are some differences in the IDriver.exe process from what was shown in Figure 4-1 for the fresh install of a Standard project.

The first operation is to obtain the product code from the database and then verify that the Setup.ilg file exists. We have already discussed the installation information registry entries made for a Standard project during a fresh install. These registry entries are covered again below where we talk about initiating a maintenance operation from the Add/Remove Programs applet. If the Setup.ilg file is missing, the installation is treated like a fresh install instead of a maintenance operation except, of course, the operation will be much faster because the files are already installed and do not have to get copied again. The need for a maintenance operation is verified by executing the `MsiGetProductInfo` Windows Installer API function to see that the application has already been installed.

Once the IDriver.exe process has detected that the application is already installed, it needs to determine if this is to be a standard maintenance installation where the end user is offered the three options of Modify, Repair, and Remove, or whether the project was created so that only an uninstallation is available. This option for Standard projects is indicated by the DoMaintenance keyword in Setup.ini. If this keyword is set to Y, the end user is offered the standard maintenance options. If this keyword is set to N, the end user can only uninstall the application. This entry is created in the Setup.ini file through the Enable Maintenance property in a Standard project. Chapter 5 covers this project property.

The second difference in how the IDriver.exe process operates for a maintenance installation is instead of calling the `MsiInstallProduct` Windows Installer API function; it calls the `MsiConfigureProductEx` API function instead. During a fresh install of a Standard project, the `MsiInstallProduct` function was called from within the `program...endprogram` block. For a maintenance install, this operation is handled through a direct call by the IDriver.exe process to an event handler function. When the DoMaintenance keyword in Setup.ini has a value of Y, the `OnMaintenance` event handler is called. However, when the DoMaintenance keyword has a value of N, the IDriver.exe process calls the `OnUninstall` event handler. These event handlers in turn call the `MsiConfigureProductEx` function with the appropriate command line. When the Windows Installer in the msiexec.exe process has performed the requested maintenance changes to the system, control is returned to the IDriver.exe process. The event handlers are covered in more detail in Chapter 8.

The completion of the maintenance operation consists of the IDriver.exe process performing any post-maintenance operations, as well as displaying any dialogs required by the installation design. IDriver.exe also reads the log file that was created during the initial application installation and performs any maintenance operations mandated by this log file. In a Standard project, this normally consists of modifying or removing the registry information that was written during the initial installation. The location of the log file is written in the registry at install time. The location of this entry in the registry is covered below in the discussion about launching a maintenance operation from the Add/Remove Programs applet.

When you launch a maintenance operation from the Add/Remove Programs applet, setup.exe is not involved and you have an environment like what is depicted in Figure 4-10.

The Add/Remove Programs applet launches IDriver.exe directly using a `/M` switch to indicate that a maintenance operation is being initiated. The location of IDriver.exe is obtained from the registry and the uninstall information that is written there when an application is first installed. Using the Developer Art installation program created in Chapter 2, the uninstallation information key created in the registry is as follows:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\
      Uninstall\InstallShield_{691BD8FA-BF60-4A36-8A0D-F02AB035193D}
```

Under this registry key, there is a value that provides the command line to the Add/Remove Programs applet for running IDriver.exe in maintenance mode. For the Developer Art application this value name and value data pair is:

```
UninstallString=C:\PROGRA~1\COMMON~1\INSTAL~1\Driver\7\INTEL3~1\
                IDriver.exe /M{691BD8FA-BF60-4A36-8A0D-F02AB035193D}
```

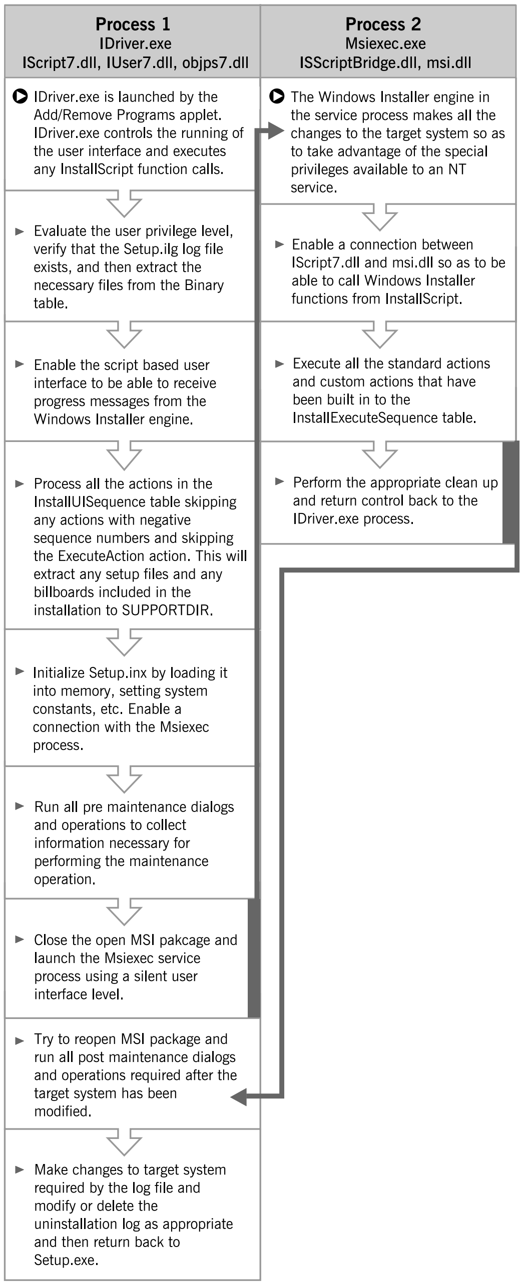

Process 1
IDriver.exe
IScript7.dll, IUser7.dll, objps7.dll
IDriver.exe is launched by the Add/Remove Programs applet. IDriver.exe controls the running of the user interface and executes any InstallScript function calls.
Evaluate the user privilege level, verify that the Setup.ilg log file exists, and then extract the necessary files from the Binary table.
Enable the script based user interface to be able to receive progress messages from the Windows Installer engine.
Process all the actions in the InstallUISequence table skipping any actions with negative sequence numbers and skipping the ExecuteAction action. This will extract any setup files and any billboards included in the installation to SUPPORTDIR.
Initialize Setup.inx by loading it into memory, setting system constants, etc. Enable a connection with the Msiexec process.
Run all pre maintenance dialogs and operations to collect information necessary for performing the maintenance operation.
Close the open MSI pakcage and launch the Msiexec service process using a silent user interface level.
Try to reopen MSI package and run all post maintenance dialogs and operations required after the target system has been modified.
Make changes to target system required by the log file and modify or delete the uninstallation log as appropriate and then return back to Setup.exe.
Process 2
Msiexec.exe
ISScriptBridge.dll, msi.dll
The Windows Installer engine in the service process makes all the changes to the target system so as to take advantage of the special privileges available to an NT service.
Enable a connection between IScript7.dll and msi.dll so as to be able to call Windows Installer functions from InstallScript.
Execute all the standard actions and custom actions that have been built in to the InstallExecuteSequence table.
Perform the appropriate clean up and return control back to the IDriver.exe process.

Figure 4-10: *Standard project run-time architecture on Windows NT/2000/XP for a maintenance installation initiated from the Add/Remove Programs applet.*

Using this command line the Add/Remove Programs applet launches IDriver.exe and passes to it the ProductCode of the application with the /M switch that has the ProductCode as its argument.

The IDriver.exe process then performs all the initialization operations, as it would do if this were a fresh install of a Standard project. Once the initialization is complete, then just as described above, the Msiexec process is launched using the `MsiConfigureProductEx` Windows Installer API.

The Windows Installer performs all the target system modifications. Then, control returns to the IDriver.exe process, where the final operations are performed and dialogs are displayed. Part of these final operations consists of reading the Setup.ilg file created during the initial installation and performing any actions indicated by this file. The location and name of this file is found under the same uninstall key in the registry as shown above. A value name value data pair under this key provides the location of the log file. For the Developer Art application this value is:

```
LogFile=C:\Program Files\InstallShield Installation Information\
                    {691BD8FA-BF60-4A36-8A0D-F02AB035193D}\Setup.ilg
```

As already discussed earlier, using the DISK1TARGET system variable you can modify this location for the log file. Remember that all system changes should be performed in the msiexec.exe process using the standard Windows Installer actions and InstallScript or native custom actions. The only operations that should be carried out in the IDriver.exe process during an installation are those related to gathering information and displaying a user interface. When this process is followed, the responsibility for modifying the target system rests with the Windows Installer. This way, you gain all the benefits provided by this technology.

Maintenance Install Using a Basic MSI Project

A maintenance operation on an application that was initially installed using a Basic MSI project has the same run-time architecture as described above for the fresh install of that application. The fresh install run-time architecture for a Basic MSI project is shown in Figure 4-7 and in Figure 4-8. The reason that a maintenance operation has the same run-time architecture is because the Windows Installer handles everything.

If the end user launches a maintenance operation by running setup.exe on the original package, all that happens is that setup.exe performs that same initialization operations as for a fresh install and then launches the client Msiexec process. The Windows Installer detects that the application is already installed and performs the appropriate actions. The same thing is true if the end user launches the maintenance operation from the Add/Remove Programs applet. The only thing that is different here is that setup.exe is not involved. When InstallScript custom actions come into play in any Basic MSI project, an IDriver.exe process is created to handle the calls to these custom actions. The run-time architecture in this case is the same as shown in Figure 4-7.

Regardless of whether InstallScript custom actions are used in a Basic MSI project, this installation type does not create a Setup.ilg file nor does it create any special registry entries other than those that are created by the Windows Installer. When you are working with a Basic MSI project, the uninstallation log is the registry itself. The Windows Installer writes many entries to the registry and these entries are located under many different keys.

There are many other installation modes that are possible when using either a Standard project or a Basic MSI project. The next section takes a look at a few of these other installation modes.

Run-Time Architecture for Other Install Modes

Up to this point we have covered the architecture for running fresh and maintenance installs using both Standard projects and Basic MSI projects. There are two other top-level actions recognized by the Windows Installer. This section takes a look at these two other install modes and relates them to the fresh install and maintenance install architectures already discussed. We also look at how localized installations are managed.

We begin this discussion with the two other top-level actions defined by the Windows Installer.

Administrative Installations

An administrative installation is not an installation in the true sense of the word. An administrative installation is meant to target a network location to which people on the network come and run the actual install of the application. During an administrative installation, no registry entries are made, no shortcuts are created, and the application cannot be launched. The only thing that takes place during an administrative installation is that any application source files that are compressed in cabinet files are uncompressed. The primary reason for uncompressing the source files for the application is so that the administrative image can be upgraded using a patch.

Standard Project

For a Standard project an administrative installation can be launched by simply passing the /a switch on the command line to setup.exe. This command line would look like the following:

```
setup /a
```

Basic MSI project

For a Basic MSI project, the end user can launch an administrative installation in one of two ways. They can do what was described above for a Standard project and pass the /a switch to setup.exe or they can pass the /a switch to the Windows Installer engine. Running the Windows Installer engine at the command line would look like the following:

```
msiexec /a <path to .msi file>
```

With a Basic MSI project, the Windows Installer engine does all the work similar to what is shown in Figures 4-7 and 4-8. The difference from a fresh install is that it is the actions in the AdminUISequence and AdminExecuteSequence tables that are executed instead of the actions in the InstallUISequence and the InstallExecuteSequence tables.

Application Advertisement

When you advertise an application, you are making it available to the end user without actually placing the source files on his or her machine until they want to use the application. An advertised application appears in the Add/Remove Programs applet and also displays a shortcut icon on the Start\Programs menu. When an application is advertised, all registry entries related to COM and file associations are made on the target machine so that the only thing that is left to do is copy the application's source files and make the non-COM related registry entries. The copying of files occurs when the end user attempts to run the application from the Start\Programs menu or tries to open a file where the application executable is registered as the extension server. Advertisement is a primary component of the Windows 2000 deployment mechanism. Chapter 3 discusses advertisement in more detail.

When an application is advertised, the actions in the AdvtExecuteSequence table are executed. There are no user interface actions implemented during advertisement. When an advertised application is first launched from the Start\Programs menu, the Windows Installer runs the installation with a basic user interface level. This means that only the actions in the InstallExecuteSequence table are executed, but the Windows Installer engine displays a built-in progress dialog during this operation. At the end of this operation, the application is launched.

Advertisement consists of two separate operations. First the application is advertised so it is made available to the end user, but the application is not actually installed. When the end user attempts to run the application that appears to be installed, the installation runs, displaying only a small progress dialog. Then the application is launched and the end user can use it. It is important to understand these two separate operations that take place when we discuss how advertisement is implemented for both Standard projects and Basic MSI projects.

Standard Project

You can advertise an uncompressed Standard project on a per-machine basis by passing to setup.exe the /j switch. When you perform this operation, the run-time architecture looks very much like what is shown in Figure 4-1 for a fresh install. The main difference is that the IDriver.exe process runs all the actions that are in the AdvtUISequence table instead of the actions in the InstallUISequence table. By default, the AdvtUISequence table has no actions inserted in it and this is the way it should remain.

After the script is initialized, the IDriver.exe process calls the undocumented event handler named `__OnAdvertisement` instead of running the `program` ...`endprogram` block as in a fresh install. The `__OnAdvertisement` event handler in turn calls the documented `OnAdvertisementBefore` and `OnAdvertisementAfter` event handlers. These two documented event handlers are no-ops by default. Between these two event handlers, a function is called that launches the msiexec.exe process to run the actions in the AdvtExecuteSequence table. These actions make the registry entries for the application and place the shortcut on the Start\Programs menu.

When the advertised application is run for the first time, the operation is completely handled by the Windows Installer and no aspects of the Standard project come into play. The Windows Installer runs the installation, as described above, with a basic user interface level so only the actions in the InstallExecuteSequence table are invoked. In the terms of a Standard project, this constitutes a silent install because the user interface sequence is not run.

By default, an advertised Standard project application cannot be installed without specific prior action by the setup developer. The InstallExecuteSequence table of a Standard project contains the OnCheckSilentInstall custom action that is inserted just prior to the LaunchConditions standard action. The OnCheckSilentInstall custom action checks if the application installation has been launched in silent mode without going through setup.exe. How this can be accomplished at the command line is discussed in the next section on Basic MSI projects. This scenario also occurs when an advertised Standard project application is first launched from the Start\Programs menu.

The OnCheckSilentInstall custom action runs only if the application has not already been installed. When it runs, it checks if the setup is script driven. If the installation is identified as script driven, the custom action returns control to the Windows Installer and the installation proceeds. A Standard project installation is only identified as script driven if it has been launched using setup.exe. If an advertised application is being launched from the Start\Programs menu, the OnCheckSilentInstall custom action sees that the installation is not script driven and calls the `OnMsiSilentInstall` event handler.

The default implementation of the `OnMsiSilentInstall` event handler aborts any attempt to run the installation of an advertised application. If you want your application to be advertised properly, you need to modify the default implementation of the `OnMsiSilentInstall` event handler. The easiest thing that you can do is to make the `OnMsiSilentInstall` event handler a no-op by removing all the code in this function. This will have the effect of allowing an advertised application to be fully installed.

If you have a Standard project where you do not want to support advertisement, then it might be a good idea to place a custom action in the AdvtExecuteSequence table to prevent the user from advertising the application. You could also place some code in the `OnAdvertisementBefore` event handler to stop the advertisement of an application, but this would be effective only if the advertisement was launched using the /j switch with setup.exe. This would not prevent the end user from advertising the application directly from the .msi file as described in the next section.

Currently, it is not possible to advertise a compressed Standard project without first running an Administrative installation to uncompress the files. It is also not possible to advertise a Standard project for the current user when you start with setup.exe. It is only possible to advertise a Standard project for all users of the machine unless the .msi file is run directly, as described in the next section.

Basic MSI Project

You can advertise an uncompressed Basic MSI project by passing the /j switch to setup.exe, just as with a Standard project. For a Basic MSI project, this switch is passed on to the Windows Installer so the architecture here looks very similar to that shown in either Figure 4-7 or Figure 4-8, depending on whether InstallScript custom actions are used. The only difference is that now the actions in the AdvtUISequence and the AdvtExecuteSequence tables are run, instead of the actions in the InstallUISequence and InstallExecuteSequence tables.

You can also advertise a Basic MSI project by passing the appropriate command line to the Windows Installer engine. An example of such a command line is:

```
msiexec /j[u|m] <path to .msi file>
```

Here the optional arguments to the /j switch indicate whether you want to advertise the application for the current user (u) or you want to advertise the application for all users of the machine (m). When you just use the /j switch without any arguments, you advertise for all users of the machine.

If you take care to make changes to the `OnMsiSilentInstall` event handler in a Standard project, you can advertise the .msi file created as part of the Standard project using the above command line. Just as with a Standard project, you cannot advertise a compressed Basic MSI project with out first performing an Administrative install.

Localized Installations

InstallShield Developer has the ability to create installation projects where the end user can select the language in which the user interface runs. It is also possible to create installation projects that display the language of the target operating system when the end user is not provided the opportunity to select the language used in the installation.

The approach used to make a particular language available for a certain installation is very different for a Standard project than it is for a Basic MSI project. However, the mechanism for deciding which language to display in the user interface is the same for both project types. This is because the multiple language functionality of InstallShield Developer is handled by setup.exe and this executable is the same for both Standard and Basic MSI projects.

If you decide at build time to offer a language selection dialog to the end user, the installation will be run in the language that the end user selects. The only problem that can arise here is if the target system does not support the selected language. In this case, the user interface will contain garbage characters. If you choose to have a language selection dialog and provide only one language, this dialog is not displayed and the installation is run in the one language that is included in the installation project. Whether a language selection dialog is to be displayed when more than one language is available is indicated in Setup.ini, as described earlier in this chapter. In this instance the following keyword and value will be found under the `[Startup]` section.

```
EnableLangDlg=Y
```

When you include a number of languages in your project but do not want the end user to select the language for the user interface, simple logic is used to determine the language to be displayed in the user interface. This logic is based on the system locale of the target operating system. If one of the included languages is the same as the system locale language, then this language will be used in the user interface. If there is no match with an included language and the language of the system locale, the user interface is displayed using the default language. There is always one language that is selected as the default language when you build an installation project.

Once a language has been selected for display in the user interface, the mechanism that is used to run the installation is as described earlier for the fresh or the maintenance installation. The next section looks at how each of the two projects makes a language available at run time.

Standard Project

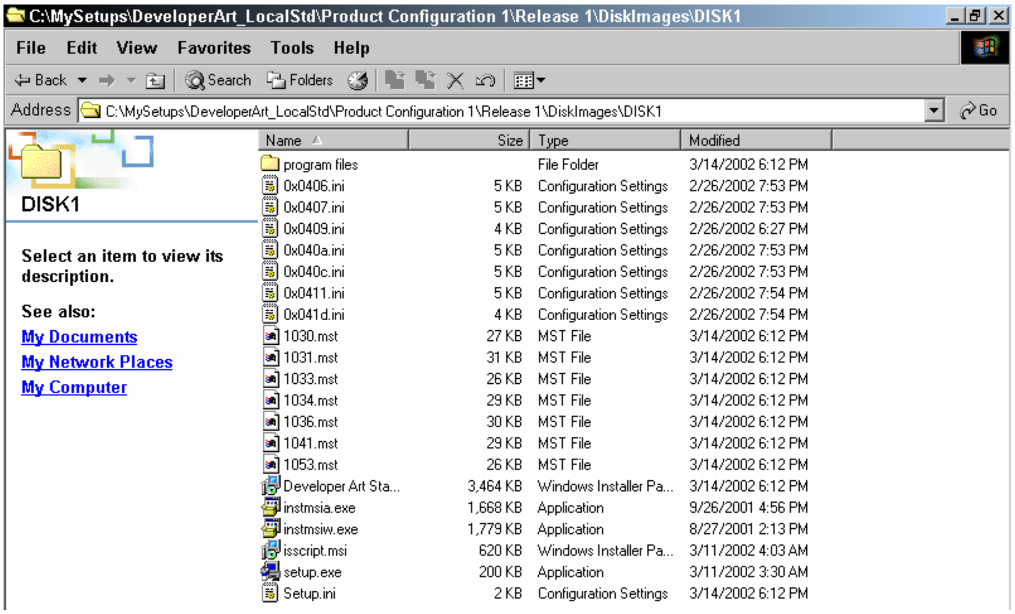
In a Standard project, the user interface is implemented using InstallScript and does not display any dialogs from the .msi file. This requires the presence of a language initialization file, a resource dynamic link library, a string table, and a transform. Chapter 3 discusses the use of transforms. The resource DLL contains all the dialog templates and the default strings in the appropriate language. The string table contains any of the custom strings that are to be displayed. Custom strings are displayed in the installation user interface or are displayed on the desktop after the application is installed. The strings in the string table can be accessed in the script that is driving the user interface for a Standard project.

As an example, you can look at a multiple language build for the Developer Art application. In this build, you should include seven languages: Danish, English, French, German, Japanese, Spanish, and Swedish. Figure 4-11 shows the Disk1 image that is created for such an uncompressed build. In this figure, there are seven initialization files each named using the hexadecimal representation of language ID for each of the seven languages that have been included in this build. The strings in these initialization files are used to display error messages and strings in the initialization dialogs that are launched at the beginning of an installation. These are the strings that might be needed before the installation's user interface is displayed. The strings from only one of these initialization files are used, and the particular one that is used is based on the language chosen by the end user in the language selection dialog.

In Figure 4-11 you also see seven transforms each named using the language ID for the language that is represented. The transforms contain the strings that are displayed by the Windows Installer while changes are made to the target system. In addition, it makes changes to the shortcuts that are to be installed so the correct language is used on the desktop for the shortcut. The tables that are modified by the transform are the ActionText, Error, Property, Shortcut, and UIText tables

Figure 4-11 shows only the language initialization files and the language transforms, but not the resource DLLs or the string tables. The resource DLLs and the string tables for each of the included languages are streamed into the Binary table at build time. The one exception is that the English resource DLL is installed along with the InstallScript engine and is not included in the Binary table.

When the end user launches the installation from setup.exe, the first thing that is done after the end user selects the language to be used is for setup.exe to apply the transform for the selected language to the database.



C:\MySetups\DeveloperArt_LocalStd\Product Configuration 1\Release 1\DiskImages\DISK1

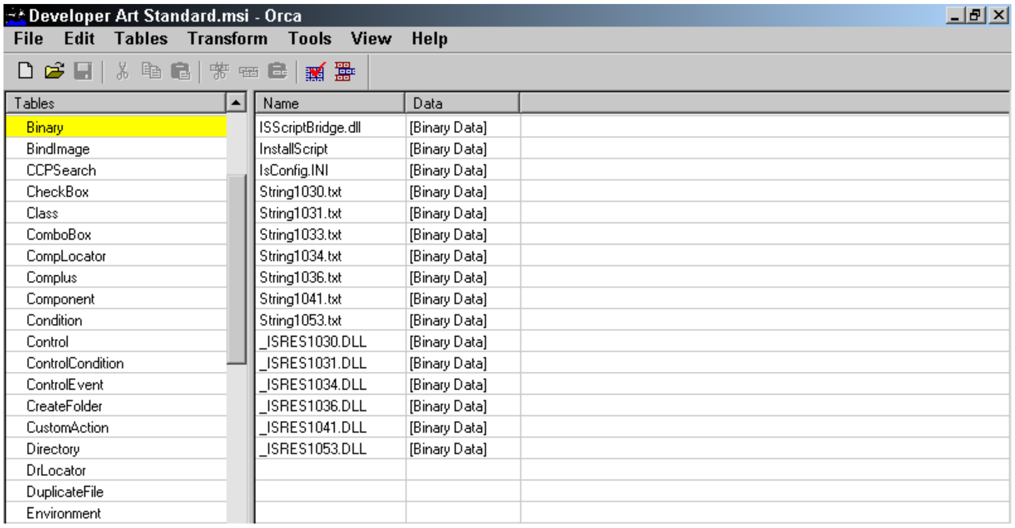
Address: C:\MySetups\DeveloperArt_LocalStd\Product Configuration 1\Release 1\DiskImages\DISK1

Name	Size	Type	Modified
program files		File Folder	3/14/2002 6:12 PM
0x0406.ini	5 KB	Configuration Settings	2/26/2002 7:53 PM
0x0407.ini	5 KB	Configuration Settings	2/26/2002 7:53 PM
0x0409.ini	4 KB	Configuration Settings	2/26/2002 6:27 PM
0x040a.ini	5 KB	Configuration Settings	2/26/2002 7:53 PM
0x040c.ini	5 KB	Configuration Settings	2/26/2002 7:53 PM
0x0411.ini	5 KB	Configuration Settings	2/26/2002 7:54 PM
0x041d.ini	4 KB	Configuration Settings	2/26/2002 7:54 PM
1030.mst	27 KB	MST File	3/14/2002 6:12 PM
1031.mst	31 KB	MST File	3/14/2002 6:12 PM
1033.mst	26 KB	MST File	3/14/2002 6:12 PM
1034.mst	29 KB	MST File	3/14/2002 6:12 PM
1036.mst	30 KB	MST File	3/14/2002 6:12 PM
1041.mst	29 KB	MST File	3/14/2002 6:12 PM
1053.mst	26 KB	MST File	3/14/2002 6:12 PM
Developer Art Sta...	3,464 KB	Windows Installer Pa...	3/14/2002 6:12 PM
instmsia.exe	1,668 KB	Application	9/26/2001 4:56 PM
instmsiw.exe	1,779 KB	Application	8/27/2001 2:13 PM
isscript.msi	620 KB	Windows Installer Pa...	3/11/2002 4:03 AM
setup.exe	200 KB	Application	3/11/2002 3:30 AM
Setup.ini	2 KB	Configuration Settings	3/14/2002 6:12 PM

Figure 4-11: *The Disk1 folder for a Standard multiple language installation project.*

Following this IDriver.exe opens the .msi file and the files are extracted from the Binary table. For our example there are a number of files that have been streamed into the Binary table. The Binary table for this example, as seen using the Orca database editing utility, is shown in Figure 4-12. Figure 4-12 shows that there are seven text files and six resource DLLs. The text files are the string tables and the one that is streamed out into a temporary directory is the one that corresponds to the language selected by the end user. The name of the text file is not changed during the extraction process. As shown, part of the file naming is the decimal language ID for the contained language.

Almost the same thing occurs with the resource DLLs. The resource DLLs also have the language ID of the supported language as part of the file name. The only resource DLL that is not in the Binary table is the one for English. This particular resource file is installed when the InstallScript engine is installed. This is discussed at the end of this chapter. When the end user selects a language other than English the resource file is streamed out of the Binary table and the name is changed to eliminate the language ID. The name of the resource DLL after it is streamed out of the Binary table is always `_ISRES.DLL`.



Name	Data
ISScriptBridge.dll	[Binary Data]
InstallScript	[Binary Data]
IsConfig.INI	[Binary Data]
String1030.txt	[Binary Data]
String1031.txt	[Binary Data]
String1033.txt	[Binary Data]
String1034.txt	[Binary Data]
String1036.txt	[Binary Data]
String1041.txt	[Binary Data]
String1053.txt	[Binary Data]
_ISRES1030.DLL	[Binary Data]
_ISRES1031.DLL	[Binary Data]
_ISRES1034.DLL	[Binary Data]
_ISRES1036.DLL	[Binary Data]
_ISRES1041.DLL	[Binary Data]
_ISRES1053.DLL	[Binary Data]

Figure 4-12: *The Binary table for a Standard multiple language installation project.*

As already explained for a fresh install of a Standard project, the files in the Binary table are streamed out to a temporary location. This temporary location is the one defined by the SUPPORTDIR system variable. On Windows 2000 this location typically has the following format:

```
%USERPROFILE%\Local Settings\Temp\{ProductCode}
```

Two other files are streamed out of the Binary table: setup.inx (called InstallScript in the Binary table) and IsConfig.INI. These files are streamed out to the same temporary location as the other files in the Binary table.

During a fresh install, the language transform is cached on the target system in a location that has the following format:

```
%SystemRoot%\Installer\{ProductCode}
```

The caching of this transform is required to make it available for maintenance operations. You do not want to perform an installation in one language and a maintenance operation in another language. The maintenance of a multiple language project is just as described in Figures 4-9 and 4-10, with the exception that the IDriver.exe process applies a language transform.

When you perform a multi-language install, the uninstall string that is written to the registry includes an additional command line argument. This additional command line argument is the language ID of the language used to perform the installation. If you install this example using German as the user interface language, the uninstall string written to the registry would have the following format:

```
UninstallString=C:\PROGRA~1\COMMON~1\INSTAL~1\Driver\7\INTEL3~1\
         IDriver.exe /M{ECA8C838-2A61-4956-83AF-4F3346C904C0} /l1031
```

The additional argument is a `"/l"` followed by the language ID of the language used to perform the installation. In this example, the language ID is 1031, which indicates that German was used to perform the initial installation.

The only difference when the end user is not provided a dialog from which to select the language to be used is that the language used is selected by the logic described above. Otherwise, there is no difference in how a language is displayed in the user interface.

Basic MSI Project

The main difference between a multi-language Standard project and a multi-language Basic MSI project that does not include any InstallScript custom actions is that, for a Basic MSI project, there is no resource DLL (_ISRES.DLL) and no string table text file required. This is because this type of project does not display any user interface that is not defined in the database tables.

For a Basic MSI project, there are still transforms for each included language that are part of the media image as shown in Figure 4-11. The content of these transforms includes the five tables described above for a Standard project and all the tables required to define the user interface in the database tables. For the Developer Art installation these additional tables are the Control, Dialog, and RadioButton tables. Depending on the user interface created for an installation there could be additional tables involved with a language transform.

When you have a Basic MSI project that uses InstallScript custom actions, you have the same situation as with a Standard project. There is a resource DLL and a string table text file for each language included in the build. These files are streamed into the Binary table and the appropriate files are streamed out from the Binary table when the end user selects the language to be displayed in the user interface. A resource DLL is required so an external dialog can be called from an InstallScript custom action.

With a multi-language Basic MSI project, the installation must be run using setup.exe. If an end user ran this type of project by just running the .msi file, the transform would not be applied and there would be no user interface for the installation. The architecture of a Basic MSI project with and without InstallScript custom actions is shown in Figures 4-7 and 4-8. The only thing that happens is that the command line that setup.exe uses to launch the msiexec.exe process includes the TRANSFORMS public property with the name of the transform to be applied. For this example, if you select German as the user interface language, the command line would look as follows:

```
TRANSFORMS=1031.mst
```

For maintenance operations, the Windows Installer automatically applies the cached transform so the language used is the same as what was used for the initial installation. If an end user accesses maintenance mode by running setup.exe again, a language selection dialog is presented, but the selection here affects only the language used in the initialization dialog. Once the maintenance operation begins, the end user will see the language that was used in the initial installation. Running the maintenance mode from the Add/Remove Programs applet avoids the language selection dialog because setup.exe is not involved.

Run-Time Handling of InstallScript

InstallScript plays a major role in running a Standard project and can also be used quite heavily in a Basic MSI package if many custom actions are required. Since InstallScript plays such an important role in running an installation, it is worth a little time to understand more about how InstallScript is handled. This discussion starts with an overview of the installation on the target system of the InstallScript engine.

Installing the InstallScript Engine

Every Standard project installs the InstallScript engine on the target machine and every Basic MSI project installs the InstallScript engine if the project uses any InstallScript custom actions. The installation of the InstallScript engine is performed using a Basic MSI project that has been modified so that there is no registration performed of the product code. This means that the InstallScript engine can be installed over and over again without ever initiating a maintenance mode operation. In fact, except for the installation of the engine from a Web site, there is no mechanism to check if the engine has already been installed. The engine is always installed from the source media. The file versioning rules of the Windows Installer prevent an older version of the InstallScript engine from replacing a newer version that may already be on the target machine.

The InstallScript engine installation package is named isscript.msi and its installation is always run silently. The InstallScript engine consists of six files named IDriver.exe, iUser7.dll, iscript7.dll, objps7.dll, _ISRES1033.DLL, and ISRT.DLL. Six components are used to install these six files and these components have NULL component codes so the Windows Installer does not know about these files after they have been installed. These files are installed to the following location:

```
C:\Program Files\Common Files\InstallShield\Driver\7\Intel 32
```

The first four of these files are COM servers and need to be registered. The isscript.msi installation package contains a special custom action that is used to make the registry entries. The built-in functionality of the Windows Installer is not used to create the COM registry entries.

The InstallScript engine cannot be uninstalled from the Add/Remove Programs applet since the Property table has set the ARPSYSTEMCOMPONENT property to a value of 1. When this property is set in an installation package it will prevent the product from being listed in the Add/Remove Programs applet. The only method for uninstalling the InstallScript engine is to do it manually.

The purpose of the files _ISRES1033.DLL and ISRT.DLL is discussed below:

_ISRES1033.DLL: This file is an English resource DLL that is used to provide the dialog template for all built-in dialogs available to a Standard project. This DLL contains all the built-in text and error messages that can be displayed during an installation. The number 1033 is the language ID for English.

ISRT.DLL: This file is a DLL that implements the built-in functions that are available in InstallScript.

After setup.exe installs the isscript.msi package it copies the above two files over to the location specified by the SUPPORTDIR system variable. During the copy of the file _ISRES1033.DLL the name is changed to _ISRES.DLL. As already discussed there is a different functionality if the installation package or program contains more than English and the end user selects a language other than English for the user interface of the installation. In the case of a multi-language install package where the end user selects a language other than English the resource DLL for the selected language is streamed out of the Binary table to the temporary location.

The Program Block and Event Handlers

The `program` block has already been discussed when covering the implementation of a fresh install with a Standard project. It is now time to take a look at the program block to better understand what actions are included (Figure 4-13). The information provided in Figure 4-13 is for background purposes only and you should not be creating your own version of the `program...endprogram` block. One reason is that this could change in later versions of the product. Also, if you were to create your own `program...endprogram` block and start placing additional functions in between the event handlers shown in this figure you would not have the advantage of all the exception handling that is incorporated in these event handlers. Remember that the `program` block is only used for the fresh install of a Standard project.

```
///////////////////////////////////////////////////////////////////////
//
//  IIIIIII SSSSSS
//    II    SS                     InstallShield (R)
//    II    SSSSSS  (c) 1996-2000, InstallShield Software Corporation
//    II        SS  (c) 1990-1996, InstallShield Corporation
//  IIIIIII SSSSSS                All Rights Reserved.
//
//
//    File Name:  Setup.rul
//
//  Description:  InstallShield script
//
//     Comments:  This is the default program block.
//
///////////////////////////////////////////////////////////////////////
#include "ifx.h"
#include "EventsConv.rul"

//Default program/endprogram block
program

    Enable(DIALOGCACHE);

    //Initialize PC Restore variables
    bIfxPCHOn = TRUE;
    bIfxPCHInitialized = FALSE;
    nIfxPCHType = REPAIR;

    ISWIPCRestoreBefore();

    ISWIOnInitInstall();

    ISWIOnCCPSearch();
    ISWIOnAppSearch();
```

Figure 4-13: *The program block as used in all Standard projects where no explicit program block is defined.*

```
    ISWIOnFirstUIBefore();

    ISWIOnMoveData();

    ISWIOnFirstUIAfter();

    ISWIOnExitInstall();

    ISWIPCRestoreAfter();

endprogram
```

Figure 4-13: *Continued.*

In the following list each of the functions that are called in the default `program...endprogram` block are briefly discussed.

ISWIPCRestoreBefore: This function handles the setting of a restore point on Windows ME and Windows XP. Restore points are created to allow end-users a choice of previous system states. Each restore point contains the necessary information needed to restore the system to the chosen state. Restore points are created before changes are made to the system in a System Restore compliant installation program.

ISWIOnInitInstall: This function initializes default installation settings. It sets the exit and help handler functions and then it calls the `OnBegin` event handler. By default the `OnBegin` event handler is a no-op. When you add code to this event handler in your script, the linking process replaces the default implementation.

ISWIOnCCPSearch: This function calls the `OnCCPSearch` event handler. By default, the `OnCCPSearch` event handler is a no-op. When you add code to this event handler in your script, the linking process replaces the default implementation.

ISWIOnAppSearch: This function calls the `OnAppSearch` event handler. By default, the `OnAppSearch` event handler is a no-op. When you add code to this event handler in your script, the linking process replaces the default implementation.

ISWIOnFirstUIBefore: This function calls the `OnFirstUIBefore` event handler. This event handler runs the user interface for the installation. When you add code to this event handler, the modified code is linked instead of the default implementation.

ISWIOnMoveData: This function calls the `ComponentTransferData` function. Its purpose is to copy files to the system and perform any other changes that have been defined such a making registry entries and creating shortcuts. There is no event handler directly called by this function but the call to the `ComponentTransferData` function brings into play all the before and after data transfer event handlers. These event handlers are discussed in Chapter 8.

ISWIOnFirstUIAfter: This function calls the `OnFirstUIAfter` event handler. This event handler runs the user interface after the installation is complete. When you add code to this event handler, the modified code is linked instead of the default implementation.

ISWIOnExitInstall: This function calls the `OnEnd` event handler. By default, the `OnEnd` event handler is a no-op. When you add code to this event handler in your script, the linking process replaces the default implementation.

ISWIPCRestoreAfter: This function marks the end of the end of the changes to the system and sets another restore point.

All the above functions perform exception handling on errors that occur and are not handled by some other means. A complete discussion of the documented event handlers that can be used by setup developers is held in Chapter 8.

This discussion is only applicable to the implementation of fresh installs using a Standard project. All other install operations have the applicable event handlers called directly by IDriver.exe. The `program...endprogram` block does not come into play with these other types of install operations.

InstallScript Custom Actions

This section examines the mechanism for running InstallScript custom actions. The mechanism requires the implementation of cross-process communication because the InstallScript engine is running in the IDriver.exe process and the custom actions are called in the msiexec.exe process. It is necessary to get the call made to an InstallScript function from the msiexec.exe process over to the IDriver.exe process so the function can be executed, and then pass the results of the function call back to the msiexec.exe process.

Figure 4-14 diagrams the flow of communication that enables the calling of InstallScript functions as custom actions. We will take a close look at this process, starting with the call to the custom action by the Windows Installer.

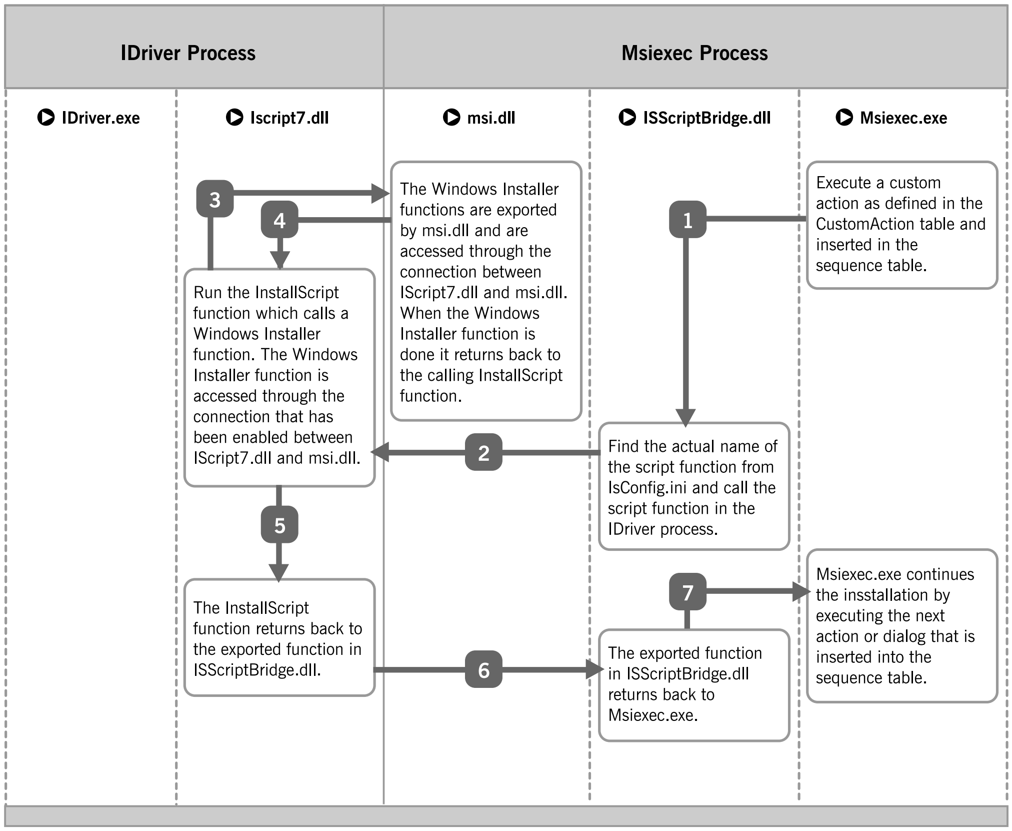
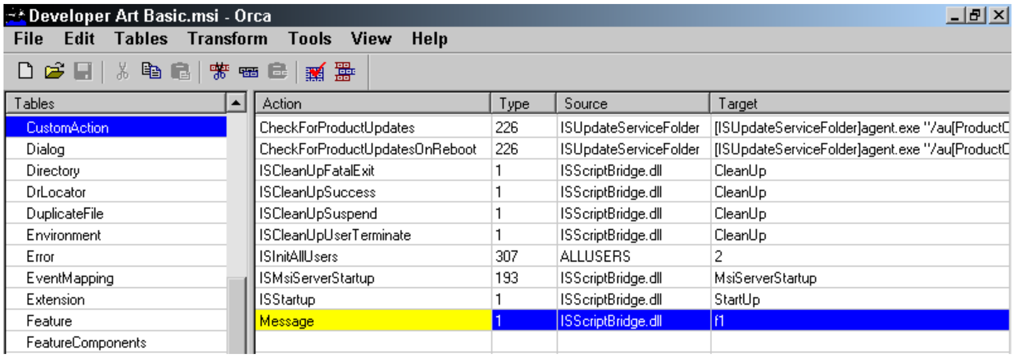

IDriver Process
Msiexec Process
IDriver.exe
Iscript7.dll
msi.dll
ISScriptBridge.dll
Msiexec.exe
Execute a custom action as defined in the CustomAction table and inserted in the sequence table.
1
Find the actual name of the script function from IsConfig.ini and call the script function in the IDriver process.
2
Run the InstallScript function which calls a Windows Installer function. The Windows Installer function is accessed through the connection that has been enabled between IScript7.dll and msi.dll.
3
The Windows Installer functions are exported by msi.dll and are accessed through the connection between IScript7.dll and msi.dll. When the Windows Installer function is done it returns back to the calling InstallScript function.
4
5
The InstallScript function returns back to the exported function in ISScriptBridge.dll.
6
The exported function in ISScriptBridge.dll returns back to Msiexec.exe.
7
Msiexec.exe continues the insstallation by executing the next action or dialog that is inserted into the sequence table.

Figure 4-14: *The calling of an InstallScript custom action.*

For purposes of discussion, we will assume that you have an InstallScript custom action and the name of the InstallScript function that implements this custom action is `InstallNTServiceMsg`. We will also assume that this custom action makes a call to one of the Windows Installer database functions. You do all the appropriate things in order to export this function, define a custom action named Message where the `InstallNTServiceMsg` function is the target, and then insert this custom action into one of the sequence tables. We are not discussing here how to create an InstallScript custom action. Chapter 11 provides a full discussion of how to create InstallScript custom actions. When you build the project, entries are made in the appropriate sequence table, in the CustomAction table, and in the Binary table.

The first thing to recognize is that the Windows Installer does not know anything about InstallScript or the scripting engine. As far as the Windows Installer is concerned, a custom action needs to be implemented using an executable, a DLL, or implemented using VBScript or JScript. As far as the Windows Installer is concerned, an InstallScript custom action is just a custom action implemented in a DLL and the name of this DLL is ISScriptBridge.dll. You can see this if you look at the CustomAction table where an InstallScript custom action is defined.


Developer Art Basic.msi - Orca
File Edit Tables Transform Tools View Help

Tables
CustomAction
Dialog
Directory
DrLocator
DuplicateFile
Environment
Error
EventMapping
Extension
Feature
FeatureComponents

Action	Type	Source	Target
CheckForProductUpdates	226	ISUpdateServiceFolder	[ISUpdateServiceFolder]agent.exe "/au[ProductC
CheckForProductUpdatesOnReboot	226	ISUpdateServiceFolder	[ISUpdateServiceFolder]agent.exe "/au[ProductC
ISCleanUpFatalExit	1	ISScriptBridge.dll	CleanUp
ISCleanUpSuccess	1	ISScriptBridge.dll	CleanUp
ISCleanUpSuspend	1	ISScriptBridge.dll	CleanUp
ISCleanUpUserTerminate	1	ISScriptBridge.dll	CleanUp
ISInitAllUsers	307	ALLUSERS	2
ISMsiServerStartup	193	ISScriptBridge.dll	MsiServerStartup
ISStartup	1	ISScriptBridge.dll	StartUp
Message	1	ISScriptBridge.dll	f1

Figure 4-15: *The CustomAction table showing the definition of an InstallScript custom action.*

Figure 4-15 shows the CustomAction table where the Message custom action is defined. The Type column shows that this is a DLL that is streamed into the Binary table. The Source column shows that the name of the DLL is ISScriptBridge.dll and the Target column shows that the function that is to be called is named f1. You might wonder how an InstallScript function named `InstallNTServiceMsg` became a function named f1. ISScriptBridge.dll has no way of knowing the names of all the possible InstallScript custom action functions that you might create. Therefore, there is a mapping mechanism employed to match up the functions exported from ISScriptBridge.dll and the InstallScript functions that are created by setup developers. ISScriptBridge.dll exports 1000 functions named f1 through f1000 which means that there is a limit in any one project of 1000 InstallScript functions that are the targets of a custom action. Actually there are two versions of ISScriptBridge.dll where if you do not have more than 50 InstallScript custom actions then the small version is used and if you have more than 50 InstallScript custom actions then the version that allows 1000 custom actions is used. This is a space saving measure.

When an InstallScript custom action is defined, the build process also defines an initialization file named IsConfig.INI that is streamed into the Binary table along with ISScriptBridge.dll. For the example in this discussion, the IsConfig.INI file has the entries as shown in Figure 4-16.

```
 [f1]
Function=InstallNTServiceMsg
[0]
0=0
```

Figure 4-16: *The contents of a typical IsConfig.INI file.*

When the Windows Installer calls the f1 function in ISScriptBridge.dll, the first thing that is done is to read the IsConfig.INI file to determine the actual name of the InstallScript function that is the real target of the custom action. The name of the function is then passed to the InstallScript engine that is loaded in the IDriver.exe process. The iscript7.dll executes the InstallScript function by accessing it in the compiled script that is always named setup.inx.

When the call to the Windows Installer database function is reached in the InstallScript code the connection that has been enabled between the IDriver.exe process and the msiexec.exe process is used to send the function call to msi.dll that is open in the msiexec.exe process. It is in the msiexec.exe process that msi.dll is loaded and it is the msiexec.exe process where the installation database is open. To access the running database, function calls have to be performed from within the msiexec.exe process. You can see this mechanism in Figure 4-14.

The results of the Windows Installer function call are returned back to the InstallScript engine in the IDriver.exe process. When the InstallScript function is finished executing, it returns back to the function in ISScriptBridge.dll where everything started. The function in ISScriptBridge.dll then returns a value to the Windows Installer and, based on this return value, the Windows Installer either executes the next action in the sequence table or it terminates the installation.

An important capability that InstallScript custom actions have that no other type of custom actions have is the ability to access the running database even from deferred mode. In Chapter 11 we will see more about what this special capability means and how it can save you extra work.

Conclusion

This chapter's main focus was how InstallShield Developer makes use of the Windows Installer engine to make changes to the installation target. The fundamental differences between a Standard project and a Basic MSI project were shown to be in how the user interface for an installation is implemented. The differences between running a Standard project and a Basic MSI project were discussed. Depending on the type of installation that is being run, any where from two to four processes are created.

The end of the chapter provided a detailed discussion of how InstallShield Developer enables InstallScript to be used for custom actions. You also learned how an InstallScript custom action, which is actually executed in a different process, can access the running database.