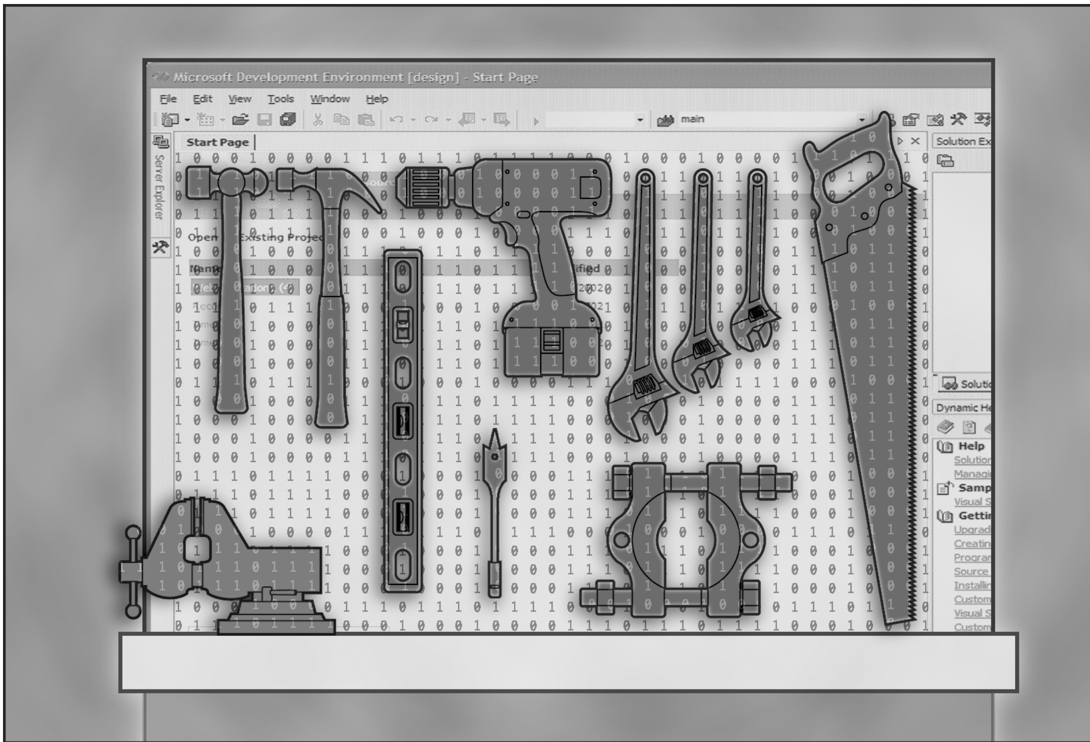
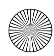
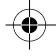
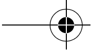
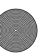
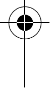
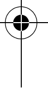
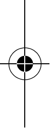
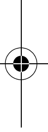
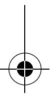
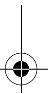
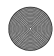
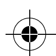
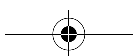
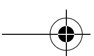
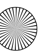
C13618747.fm Page 395 Friday, January 10, 2003 3:54 PM

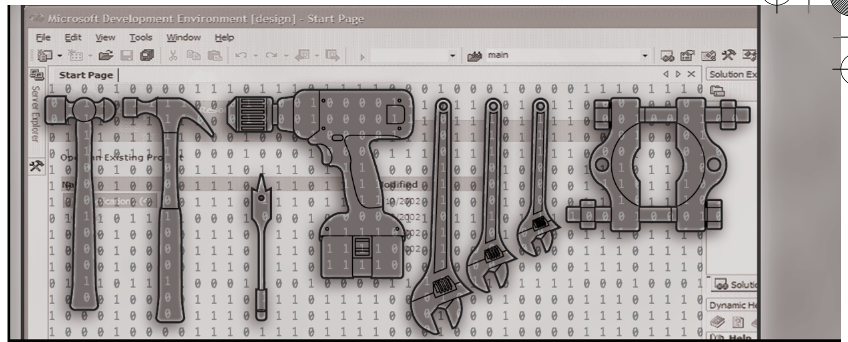
Part III

Deployment, Help, and Advanced Projects





13

Designing Setup Projects

When creating a new application, the last thing a developer usually thinks about is how to get the application onto the user's computer. In this chapter, we'll explore the tools in Microsoft Visual Studio .NET that help you create setup files for use with Windows Installer.

Microsoft Windows Installer (MSI) Background

In the early days of Microsoft Windows development, an application rarely consisted of more than a single executable file and maybe a DLL or two, so installing the application was as simple as copying the files from a floppy disk onto the computer. Changes to the system settings were rare, and when necessary they were simply a few changes to the win.ini and system.ini files in the C:\Windows folder.

Much has changed since those days; preparing even the simplest of applications so the user can run it on modern versions of Windows requires many changes to the user's computer. The setup process for a modern Windows application must modify the system registry to associate files with your application so when the user double-clicks on the file it automatically opens. It must place COM DLLs in various locations on disk and register them, causing changes to the system registry. It must modify the Start menu so the user can conveniently find and run your application. In the case of Web applications and XML Web services, installing an application on a Web server requires not only placing the files on the server's hard drive but also configuring Microsoft Internet Information Services (IIS) to serve the Web application to the user.

Since those early days of Windows development, countless setup development tools have become available. One of the most popular ones, which

C13618747.fm Page 398 Friday, January 10, 2003 3:54 PM

Microsoft used, was called Acme Setup; it was used in programs such as Microsoft Office 95 and Microsoft Visual Studio 6.0. Acme Setup and many of the other setup technologies got the job done, but they were complicated to develop for and didn't provide the user with a consistent experience from one application to another. These setup technologies could also be dangerous to the user's computer. If a problem occurred during installation, such as an error in installing a component, or if the user canceled the installation, the state of the computer would become unstable; some components or registry settings would be left behind or incorrectly removed.

To make developing setup programs easier and to solve the problems associated with existing setup programs, the Office product group set out to develop a new type of setup program. This technology is called Windows Installer. When a Windows Installer setup program is built, a file with the .msi extension is created; the user can double-click on this file to start installing a program. The files that make up the program to be installed can be either compressed and stored within the .msi file or stored loosely (on a distribution medium such as a floppy disk, CD, or DVD that's separate from the .msi file).

Because logic is built into Windows Installer for handling the system registry, COM objects, and (with version 2.0 of Windows Installer, the same version used by Visual Studio .NET and the .NET Framework) .NET components and .NET Web applications, installing these components is easy. Windows Installer also takes care of mundane setup chores such as making sure the computer has enough disk space, and it creates an entry in Control Panel's Add Or Remove Programs applet for uninstalling the program. Lastly, Windows Installer is transactional, which means the state of the computer is preserved when you install a component or a registry key. If a problem occurs during setup, a rollback is performed that restores the computer to the state it was in before setup was started.

Creating Custom Installation Projects

With the tools in Visual Studio .NET, you can easily create a setup project that, when built, generates an .msi file. You can find the templates for creating a setup project in the New Project dialog box, by selecting the Setup And Deployment Projects node in the Project Types tree. The Setup Project type is used to install client software, such as a .NET or Win32 program, onto a user's computer, and the Web Setup Project type is used to install a Web application or Web service onto a server computer. The Merge Module template (discussed later in this chapter) is used to create setup project components.

If you were to add a setup project to an existing solution and then choose Build | Build Solution to build your solution, the setup project wouldn't build because in the Configuration Manager dialog box the setup project isn't selected by default to build. Setup projects can take a while to build, and because you usually don't need to recompile the .msi file each time you want to debug a project, not building the setup project each time you compile the solution saves you some time. When you're ready to test your setup project, you can right-click on it in Solution Explorer and choose Build or you can select the Build check box in the Configuration Manager dialog box. If you're creating a setup project in a new solution file, the Build check box is selected by default because no other projects are in the solution.

Once a setup project has been added to a solution, you can choose among six editors to build your setup project: File System, Registry, File Types, User Interface, Custom Actions, and Launch Conditions. You can use any of these editors to configure how your software is installed onto a user's computer. You can display any of these editor windows by selecting a setup project in Solution Explorer and then clicking the appropriate button on the command bar or right-clicking on the project in Solution Explorer and choosing View and then the editor.

File System Editor

You use the File System editor to graphically indicate where files that make up your software project should be placed on disk when the .msi file is installed. In this editor, you can add folders and files compiled by a project to create a directory structure that's logical for your application.

Specifying an Installation Folder

To install your program, you must create the directory structure that will contain the program's files. Many default installation folders are available in the File System editor, giving you a starting point for a directory structure. To add a file to a folder, right-click on the appropriate folder, point to Add, and then select one of the file types—Folder to create a subfolder, Project Output to add a file generated by another project in the solution, File for a file on disk, or Assembly for an assembly file. Visual Studio .NET defines the following folders that you can add files or folders to:

- **Common Files Folder** For files that are common among all programs installed on the computer. This folder can be found at C:\Program Files\Common Files.

- **Fonts Folder** For all the font files installed on the computer. The default location of this folder is C:\Windows\Fonts.
- **Program Files Folder** The folder where all programs installed on the computer should be stored. It can be found at C:\Program Files.
- **System Folder** For storing operating system components. This folder should be modified only in the rarest of situations. It can be found at C:\Windows\System32.
- **User's Application Data Folder** The folder where applications can store data files that the user shouldn't manipulate. It can be found at C:\Documents and Settings*username*\Application Data.
- **User's Desktop** For all the items shown on the user's desktop. The default location for this folder is C:\Documents and Settings*username*\Desktop.
- **User's Favorites Folder** For links to favorite items. The default location for this folder is C:\Documents and Settings*username*\Favorites.
- **User's Personal Data Folder** For documents that the user creates. This folder can be found at C:\Documents and Settings*username*\My Documents.
- **User's Programs Menu** For shortcuts to programs that will be shown on the Start menu. This folder can be found at C:\Documents and Settings*username*\Start Menu\Programs.
- **User's Send To Menu** For Send To menu items, which you can see by clicking on a file in Windows Explorer and selecting the Send To menu. This folder can be found at C:\Documents and Settings*username*\SendTo.
- **User's Start Menu** For Start menu items. This folder can be found at C:\Documents and Settings*username*\Start Menu.
- **User's Startup Folder** For all programs (or shortcuts to programs) that will run when the user logs in to the operating system. This folder can be found at C:\Documents and Settings*username*\Start Menu\Programs\Startup.
- **User's Template Folder** For templates to create new files. This is the source folder where the items on the New menu of the desktop's shortcut menu are located. This folder can be found at C:\Documents and Settings*username*\Templates.

C13618747.fm Page 401 Friday, January 10, 2003 3:54 PM

- **Windows Folder** For operating system files. A typical location is C:\Windows.
- **Global Assembly Cache Folder** For the computer's global assembly cache (GAC). Any files placed in this folder are accessible by the .NET Framework for all users of the computer.
- **Application Folder** The folder where you should store most of the files and project output you're installing onto the user's computer. This folder defaults to C:\Program Files*Manufacturer**ProductName*, where *Manufacturer* and *ProductName* are the property values in the Properties window when the setup project is selected in Solution Explorer.
- **Web Application Folder** A folder that's available only if the setup project is a Web Setup Project. Items added to this folder are installed in the IIS virtual folder and are available (security settings permitting) to users of your Web server.

You can also create a new folder on the computer that's not a child of any of the default folders. To do this, right-click on the File System On Target Machine node in the File System editor and choose Add Special Folder | Custom Folder. The name you enter for the folder is not the path for the folder that will be created on the computer—it's for display purposes in the File System editor only. You set the folder path by selecting the newly created custom folder and typing the path to create in the *DefaultLocation* property in the Properties window. However, be careful when you create a custom folder and give it a hard-coded path; if you enter a disk drive that's not available on the computer, an error is generated. Later in this chapter, you'll see how you can set the path of the folder dynamically at installation time.

Project Output

Once you create a program and the directory structure to hold the program, you need a way to add the files to the File System editor. You could build your code project and then manually select each file generated by the project and add them to the File System editor. However, this approach can lead to problems. For example, if you switch the project type from Debug to Release, you must manually modify the setup project to make sure the correct build version of the files is copied into the setup project. Another problem is that you might inadvertently omit a file or add a file that is not needed, causing the program to not function properly or causing the .msi file to be bigger than it needs to be. To make selecting files to install for a project easier, you can include the project output in a setup project.

C13618747.fm Page 402 Friday, January 10, 2003 3:54 PM

To add a project's output to the File System editor, right-click on the folder that should contain the output and choose Add | Project Output. The Add Project Output Group dialog box opens and lists the projects that generate output and lists output types you can add to the setup project. The project output types you can add to a setup project are:

- **Documentation Files** This type adds any files generated by IntelliDoc to the File System editor. IntelliDoc files can be generated only from C# projects, so this option appears only if the project selected in the Project box is a C# project.
- **Primary Output** This group contains the DLL or EXE file that the project creates when compiled. You must add this output to the File System editor to enable the user to run your program.
- **Localized Resources** If you add this output type to the File System editor, all resource satellite DLLs are copied into the selected folder.
- **Debug Symbols** This output type contains all the debug symbols, such as .pdb files, that are used to debug the application. If you want users of your application to be able to debug problems, you should add this output type to the File System editor. However, placing debugging symbols on the user's computer increases the size of the .msi file. In addition, you make it easier for people to reverse-engineer your application and possibly gain access to your intellectual property.
- **Content Files** This output type includes all files within the selected project that were added as content files.
- **Source Files** This project output type includes all the source files used to build a project.
- **Built Outputs** This output type is available only if the selected project is another setup project. Adding this output type to a setup project allows you to install an .msi file onto the installer's disk.

When a project's output is added to a setup project, Visual Studio .NET automatically scans the output and adds to the setup project any files (such as assemblies, unmanaged DLLs, or type libraries) that the project is dependent on. When the setup project is compiled, these dependent DLLs are packaged up into the .msi file and installed alongside the code that's dependent on those DLLs. These dependent DLLs appear in the File System editor alongside the file or project output that is dependent on them, and they're added to Solution Explorer underneath the Detected Dependencies folder of your setup project.

You can control which dependencies are installed on the user's computer by excluding a dependency; you simply right-click on a dependency file and choose Exclude. There can be many reasons for excluding a file, the most common of which is that the user has the dependency file on her computer and therefore you don't need to package that file into the .msi file. In addition, when a project's output is added to the File System editor, a project dependency is created from the setup project to the project generating output. As a result, when a solution build is started, the project, which has its output in the setup project, is built before the setup project is compiled, which ensures that the files in the setup project are up-to-date.

Have Your Wizards Stopped Working?

When you test your setup project to make sure it installs and uninstalls properly, you might find that some programs have stopped working—especially after you uninstalled the .msi file. This problem is common when you're building a setup program for wizards or add-ins. Suppose you reference the assembly *VSLangProj* within your add-in project. When you build the setup project for the add-in, the setup project sees that you referenced the *VSLangProj* assembly and automatically adds it to the setup project as a dependency. Also, because *VSLangProj* is a Primary Interop Assembly (PIA) for the type library VSLangProj.tlb, the type library is added to the setup project as a dependency. This last file is where you can run into trouble. When COM objects (including type libraries) are added to the File System editor, they're automatically set to be registered when installed.

If the .msi file for your add-in or wizard is uninstalled, Windows Installer removes all the files that it installed on the system. Because the VSLangProj.tlb type library is being uninstalled, it also unregisters itself as a type library. Other components, such as Visual Studio .NET wizards, use this type library, and if the type library is not registered, the wizards cannot run.

To fix this problem and prevent it from recurring, you can scan your setup project after adding a new project output to the File System editor, to look for components added as dependencies that aren't part of your project. If you find such a file, right-click on that file and choose Exclude, or if you know that the dependency is located within a merge module, add the merge module to your setup project. This ensures that the component is properly installed on the computer. (*VSLangProj* is not in a merge module, so this is not an option for a wizard or an add-in.)

C13618747.fm Page 404 Friday, January 10, 2003 3:54 PM

If you've already uninstalled an .msi file that contains a dependency that shouldn't have been installed and you want to repair your computer, you have a few options. The first is to run the repair option of the application that has stopped working. If the application is Visual Studio .NET, the repair process will be lengthy and you might not want to go through it. An alternative way of fixing the problem is to find and then manually reregister the type libraries and COM objects that were unregistered. You can do this only if you have the necessary tools. The third option is to create a throwaway setup project, add the necessary components (such as VSLangProj.tlb) to the setup project, and then build and install that setup project. This causes the file to be registered, and as long as you do not uninstall this .msi file, everything should once again work fine.

Registry Editor

With the Registry editor, you can point and click your way to creating entries in the system registry when the .msi file is installed. During installation, the registry keys and values you create within this editor are copied into the system registry, mirroring the structure you create. You can see an example of the Registry editor being used to help set up an add-in when you run the Add-in Wizard. The purpose of a setup project being added to a solution when you run this wizard is not to install files (although the setup project also helps do that) but to create the registry keys necessary for Visual Studio .NET to find, load, and run your add-in.

You'll notice that the Registry editor window closely resembles the Registry Editor program (regedit.exe) that's installed with Windows. Just as you can edit the system registry using regedit.exe, you can edit the registry using the Registry editor, except that registry settings declared in the Visual Studio .NET Registry editor window are created at install time.

If you use a setup project to install a COM object created with the C++ programming language, you must decide whether to define the registry keys for that COM object within the Registry editor or let the COM object register itself during installation. When the output of a COM object project is added to the File System editor, the *Register* property in the Properties window for that output is set to *vsdrpCOM*, which means the COM object knows how to register itself, and the *DllRegisterServer* and *DllUnregisterServer* methods are called on installation or uninstallation of that object. However, if a COM object registers itself, the keys it creates aren't included in the transactional feature of Windows

C13618747.fm Page 405 Friday, January 10, 2003 3:54 PM

Installer. If installation fails, the *DllUnregisterServer* method is called to try and roll back the registry key creation; if that fails, some registry entries might be left behind. On the other hand, creating the registry entries for a COM object can be a tedious, error-prone chore. If all you need to do is copy the entries in an .rgs file to the Registry editor, it isn't a problem, but you must create multiple registry keys and values for every interface defined by that COM object so the proxy and stub DLL for that interface are set up correctly. The choice is yours, but you should consider the options carefully.

The User/Machine Hive

The Registry editor lets you modify the registry settings for the four main registry root sections, or hives: HKEY_CLASSES_ROOT, HKEY_CURRENT_USER, HKEY_LOCAL_MACHINE, and HKEY_USERS. However, the Registry editor for an .msi setup project has one additional node that's not a root key within the system registry: the User/Machine Hive key. This key is used to conditionally modify the HKEY_LOCAL_MACHINE key or HKEY_CURRENT_USER key and is dependent on an option the user selects when installing an .msi file. If you create a setup project and then build and install the resulting .msi file, the second page of the setup user interface appears as shown in Figure 13-1. On the bottom left of this dialog box are two options, Everyone and Just Me. If the user selects Everyone, all the registry keys you create in the User/Machine Hive key are placed in the system registry under the HKEY_LOCAL_MACHINE key. If the user selects Just Me, all these settings are placed in the HKEY_CURRENT_USER key of the system registry. If the person running the .msi file has reduced permissions, such as Guest, these two option buttons are not displayed and the setting defaults to Just Me.

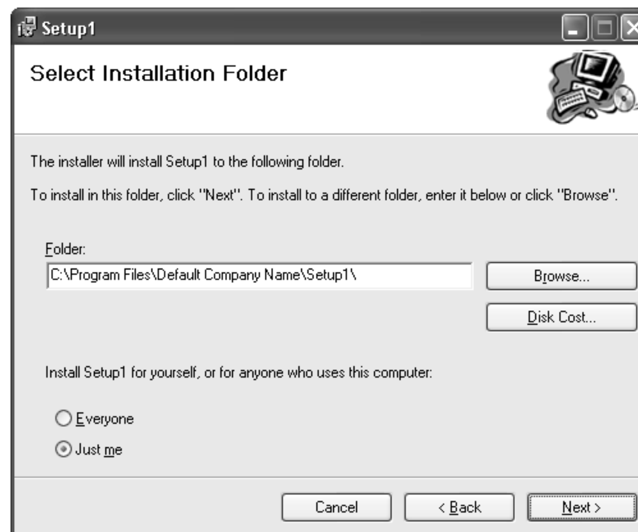


Figure 13-1 The Select Installation Folder page of an .msi setup file

Installer Properties

Although the Registry editor provides an easy-to-use, point-and-click way to create registry settings for your program, the registry modifications you make are static. That is, the key names, value names, and data values you enter are copied into the registry during setup exactly as you've typed them. However, at times you'll need to create a registry key with a name or value that's dynamic, reflecting the state of the computer when the user installs an .msi file. Such dynamic values are known as *installer properties*. Using an installer property in the Registry editor is as simple as placing the installer property name within square brackets; when the .msi file is run, Windows Installer notices these installer properties and replaces them with the appropriate values.

Table 13-1 lists the most commonly used installer properties of the nearly 200 that are available. New ones are being added with each new version of Windows Installer. You should consult MSDN for an up-to-date listing of the available installer properties. As an example, to place the date on which the user installed the .msi file into the registry, you use *[Date]* as the registry key name, value name, or value. You can combine installer properties by placing them next to one another, and you can add string data where appropriate. For example, the value *[Time]-[Date]* is expanded to create the value *17:54:35-11/10/2002* for November 10, 2002, at 5:54:35PM. You can see one of these installer properties being used when you first create a setup project. If you look in the Registry editor of a setup project, you can see that the keys HKEY_LOCAL_MACHINE\Software\[Manufacturer] and HKEY_CURRENT_USER\Software\[Manufacturer] are automatically created for you. These keys are where you can place data specific to your company's software; the token *[Manufacturer]* expands into the name of your company, which you can enter as the *Manufacturer* property in the Properties window when the setup project is selected in Solution Explorer.

Table 13-1 Commonly Used Installer Properties

Installer Property	Description	Example
AdminToolsFolder	Folder where tools to administer the computer are stored.	C:\Documents and Settings*username*\Start Menu\Programs\Administrative Tools\
AdminUser	This value is 1 if the installing user has administrator privileges, 0 if not.	1
AppDataFolder	Folder where application-specific data is stored.	C:\Documents and Settings*username*\Application Data\

C13618747.fm Page 407 Friday, January 10, 2003 3:54 PM

Table 13-1 Commonly Used Installer Properties *(continued)*

Installer Property	Description	Example
ARPCONTACT	The name of the technical support contact person. This value is set using the Author property in the Properties window when the setup project is selected.	Default Company Name
Author	The author of the installer. The value of this property is set in the Properties window when the setup project is selected in Solution Explorer.	Default Company Name
CommonAppDataFolder	The folder shared by all users for storing application-specific data.	C:\Documents and Settings\All Users\Application Data\
CommonFilesFolder	The folder where shared software components are stored.	C:\Program Files\Common Files\
ComputerName	The network name of the computer.	CRAIGS4000
Date	The date on which the .msi file is installed.	11/10/2002
DesktopFolder	The folder for installing user's desktop items.	C:\Documents and Settings*username*\Desktop\
FavoritesFolder	The folder where Internet Explorer favorites are stored.	C:\Documents and Settings*username*\Favorites\
FontsFolder	The folder where fonts are stored.	C:\Windows\Fonts\
Intel	If setup is running on a computer with one or more Intel processors, this value is the processor class being used: 4 for a 486, 5 for a Pentium, 6 for a P6, and so on.	6
LocalAppDataFolder	The folder for nonroaming, application-specific user data.	C:\Documents and Settings*username*\Local Settings\Application Data\
LogonUser	The user name of the person running the .msi file.	Craig Skibo
Manufacturer	The name of the company that created the .msi file.	Default Company Name
MyPicturesFolder	The folder where user images are stored.	C:\Documents and Settings*username*\My Documents\My Pictures\

Table 13-1 Commonly Used Installer Properties *(continued)*

Installer Property	Description	Example
MsiNTProductType	The type of the operating system installed. A value of 1 means that the computer is a workstation, 2 means that the computer is a domain controller, and 3 means it's a server.	1
NetHoodFolder	The Network Neighborhood folder.	C:\Documents and Settings*username*\NetHood\
PersonalFolder	Folder where user documents are stored.	C:\Documents and Settings*username*\My Documents\
PhysicalMemory	The amount of memory, in megabytes, on the computer where the .msi file is being run.	384
PrintHoodFolder	The folder where printers are installed.	C:\Documents and Settings*username*\PrintHood\
Privileged	This value is 1 if the installation is performed under elevated user privileges.	1
ProductID	The serial number entered in the Serial Number edit box in the User Information dialog box (described later in this chapter).	111-7000000
ProductName	The name of the product being installed. You set this value by changing the ProductName property in the Visual Studio .NET Properties window when the setup project is selected.	Setup1
ProductVersion	The version of the .msi file being installed. You set this value in the Version property in the Properties window when the setup project is selected in Solution Explorer.	1.0.0
ProgramFilesFolder	The Program Files folder.	C:\Program Files\
ProgramMenuFolder	The folder where Start menu program shortcuts are stored.	C:\Documents and Settings*username*\Start Menu\Programs\
RecentFolder	The folder where shortcuts to recently used documents are stored.	C:\Documents and Settings*username*\Recent\

C13618747.fm Page 409 Friday, January 10, 2003 3:54 PM

Table 13-1 Commonly Used Installer Properties *(continued)*

Installer Property	Description	Example
RemoteAdminTS	This value is 1 if the computer has terminal services installed and configured.	1
ROOTDRIVE	The drive on which to install the program.	C:\
ScreenX	The width, in pixels, of the user's primary monitor.	1024
ScreenY	The height, in pixels, of the user's primary monitor.	768
SendToFolder	The folder containing items shown on the context menu when you right-click on a file in Windows Explorer and choose Send To.	C:\Documents and Settings*user-name*\SendTo\
ServicePackLevel	The current service pack version installed on the computer.	1
ServicePackLevel-Minor	The minor version number of the service pack installed.	0
SourceDir	The folder containing the .msi file.	C:\Documents and Settings*username*\My Documents\Visual Studio Projects\Setup1\Debug\
StartMenuFolder	The folder where Start menu shortcuts are stored.	C:\Documents and Settings*username*\Start Menu\
StartupFolder	The folder containing links to programs that are started when the user logs in to the operating system.	C:\Documents and Settings*username*\Start Menu\Programs\Startup\
SystemFolder	The Windows system folder.	C:\Windows\System32\
SystemLanguageID	The locale identifier (LCID) of the operating system.	1033
TARGETDIR	The folder where the setup project is being installed.	C:\Program Files\Default Company Name\ Setup1\
TempFolder	The folder for temporary files.	C:\ Documents and Settings\ *username*\Local Settings\Temp\
TemplateFolder	The folder where templates are stored. Templates are the items shown on the New menu when the context menu of the desktop is displayed.	C:\Documents and Settings*username*\Templates\

Table 13-1 Commonly Used Installer Properties *(continued)*

Installer Property	Description	Example
Time	The time when the .msi file is being installed, in the format HH:MM:SS.	17:54:35
UserLanguageID	The locale identifier (LCID) in use by the user.	1033
USERNAME	The logon name of the user installing the .msi file.	craigs
VersionNT	The version of operating system being used if the operating system is 32-bit NT class.	501
VirtualMemory	The amount of memory, in megabytes, assigned to the virtual memory page.	576
WindowsBuild	The build number of the operating system.	2600
WindowsFolder	The folder in which the operating system is installed.	C:\Windows\
WindowsVolume	The hard disk drive on which Windows is installed.	C:\

You can use these installer property values not only in the Registry editor but also in the File System editor. The previous section described how you can define custom folders that are created when the .msi file is installed, but the path to those folders was hard-coded. Using installer properties, you can specify that the path of a custom folder be determined at install time.

Suppose you need to create a folder for storing application-specific data. The installer property *AppDataFolder* is typically set to the value *C:\Documents and Settings*\username*Application Data*\, which is the folder where user-specific data for a program, such as configuration options, should be stored. The installer property *ProductName* is set to the name of the product being installed, which defaults to the name of the setup project. You could set the custom folder's location to point to the path C:\Documents and Settings*username*\Application Data*ProductName*, but if the user installs the operating system to the D drive or changes the application data path, this won't be the correct location to store data. You can combine the installer properties *AppDataFolder* and *ProductName* and set the *DefaultLocation* property for the folder to create within the Properties window into *[AppDataFolder][ProductName]*. This automatically creates the folder C:\Documents and Settings*username*\Application Data\Setup1 (where the name of the setup project is Setup1).

C13618747.fm Page 411 Friday, January 10, 2003 3:54 PM

File Types Editor

Of all the technologies built into Windows to help users get started using their computers, file associations are probably the most overlooked. Back in the days of MS-DOS, if you wanted to view or edit a data file, you had to know which application could be used to edit that file, and then you had to know how to start that application to view it. With Windows, all you need to know is how to double-click, because when you double-click the icon in Windows Explorer, the application associated with that file is automatically run and the data file is loaded. You create associations by creating a set of system registry keys and values that link the extension of a data file to the program that views or edits that file. You can use the Registry editor of a setup project to define your file associations, but this can be complicated. The File Types editor in Visual Studio .NET lets you easily define your file associations.

Suppose you create a new way of storing image data in a compressed format that's better than any other image format available. You've also created a .NET program named MyImageViewer to make viewing, printing, and editing that file format possible. If the user of this file format wants to print the image, she can open your viewer application, choose File | Open, browse to the image (which has the file extension .myimage), and then choose File | Print to print the image. Or, she can let Windows handle all the work for her using a file association. For the purposes of this example, we'll leave the theories about image file formats to other books and use a bitmap file, renamed to have the .myimage extension, as our file format.

To create a .myimage file association within a setup project, first open the File Types editor for the setup project by right-clicking the setup project in Solution Explorer and choosing View | File Types. Right-click on the File Types On Target Machine node, and select Add File Type. This creates a file association for the Open verb. A verb is an action you take against the file; the default verb (denoted within the File Types editor using boldface) is the action performed when the user double-clicks on the file in Windows Explorer. Other common verbs are Print and Edit. For our example, we'll add both of these.

To add the Print verb, right-click on the file type node you just created, choose Add Action, and then type **&Print**. The verb name is preceded by an ampersand because this text will be displayed to the user when a .myimage file is right-clicked in Windows Explorer and the P key will be the shortcut key for printing. Next, you set how the verb will tell your program that it has been invoked. Select &Print and, in the Properties window, type the command-line argument for the Print verb in the *Arguments* property. For this example, the command-line argument is *–print "%1"*. The *–print* value is a command-line switch that tells your program it should print a file. Windows Explorer replaces

the *%1* token with the file path of the image that is to be printed. This token should be surrounded by double quotes. If it isn't and the file path has an embedded space, two or more strings will be passed as the filename to the program's command line arguments, not just one, thereby confusing the program that handles the verb. Next, in the Properties window, type the name of the verb, **Print**, in the *Verb* property. For the purposes of this example, you should also repeat the process to create another verb, using the verb Edit in place of Print where appropriate.

Now you need to tell the setup project which program to run when the user selects one of these verbs. Select the file type node you created underneath the File Types On Target Machine node, and then open the Properties window. The *Command* property specifies the program that will run when the verb is invoked; you can set the target of the verb to any file that's been added to the File System editor, including project output such as the primary output. To specify an extension that is associated with the program, type one or more extensions, separated by semicolons, in the *Extensions* property. For this sample, type **myimage**; a period preceding the extension isn't required. If you want to use a custom icon for your file format when the file is viewed in Windows Explorer, you can add an icon to the File System editor and then browse to that icon using the *Icon* property.

The last step is to modify your program to accept the command-line parameters passed to the program. If your program is a Windows Forms application and the main form in the program is called Form1, you can add the following constructor to the *Form1* class:

```
public Form1(string []args)
{
    InitializeComponent();
    if(args.Length == 1)
    {
        pictureBox1.Image = System.Drawing.Bitmap.FromFile(args[0]);
        this.Text = this.Text + " - " +
                    System.IO.Path.GetFileName(args[0]);
    }
    else if (args.Length == 2)
    {
        //The two command line switches that are recognized:
        string printCommand = "-print";
        string editCommand = "-edit";
        if(System.String.Compare(printCommand, args[0], true) == 0)
        {
            //We were asked to print the image. Load the image, then use
            // the PrintDocument class to print
            pictureBox1.Image = System.Drawing.Bitmap.FromFile(args[1]);
            PrintDocument printDocument = new PrintDocument();
            printDocument.PrintPage += new
                PrintPageEventHandler(printDocument_PrintPage);
```

C13618747.fm Page 413 Friday, January 10, 2003 3:54 PM


```
            printDocument.Print();
            System.Diagnostics.Process.GetCurrentProcess().Kill();
        }
        else if(System.String.Compare(editCommand, args[0], true) == 0)
        {
            //We were asked to edit the image.
            //Spawn off to MSPaint to edit:
            System.Diagnostics.Process.Start("mspaint.exe", "\""
                                             + args[1] + "\"");
            System.Diagnostics.Process.GetCurrentProcess().Kill();
        }
    }
}
```

Next, change the *Main* function to the following:

```
static void Main(string []args)
{
    Application.Run(new Form1(args));
}
```

When the Edit or Print verbs are invoked, the command line to the program is –edit *filename* or –print *filename*. This code examines the parameters, and if *–print* or *–edit* is specified, it takes the appropriate action, either printing the image or calling to mspaint.exe to display the image passed on the command line, and then it exits. If neither verb is specified and the user wants to view the image, the image file is loaded and displayed on the form. The MyImageViewer sample contains the complete source code for a .myimage viewer and a setup project that includes the settings to register a file extension.

User Interface Editor

When an .msi file is installed, a setup wizard walks the user through the install process. The wizard's dialog boxes do little more than tell the user which program he's installing, ask for the name of the folder on disk in which to install the program, and provide feedback during installation. Using the setup tools built into Visual Studio .NET, you can add dialog boxes that ask for more information about how to configure your program's installation.

You can customize any dialog box within a setup project by modifying two properties in its Properties window: *BannerText* and *BannerBitmap*. *BannerText* specifies the text in the banner at the top of the dialog box, which describes that step of the setup wizard. *BannerBitmap* specifies a bitmap file to show along the top of the dialog box; you must add this bitmap to the File System editor before you can browse to it, and it must be 496 pixels wide and 68 pixels high. The bitmap can look any way you want, but keep in mind that the value of the *BannerText* property will appear on top of the bitmap; you should use a color that will allow this text to be visible.

C13618747.fm Page 414 Friday, January 10, 2003 3:54 PM

In the User Interface editor, you'll notice two branches of a tree, Install and Administrative Install. The Install branch is where you do most of the work to design the user interface of a setup project. The dialog boxes shown in this branch make up the user interface that most users see; they walk users through the steps of setting up your application. The Administrative Install branch, as its name suggests, is for system administrators. If a network administrator runs an .msi file with the *–a* switch on a command line, such as *msiexec.exe –a msifile.msi*, the files contained in the .msi file are installed so the users of the network can perform a network installation of the program. Only a subset of the dialog boxes available in the Install branch can be used in the Administrative Install branch. In most situations, you don't need to make changes to the Administrative Install branch of the User Interface editor; Windows Installer handles all the details of an administrative install for you.

Splash Screen

The purpose of a splash screen page is simply to display an image to users that identifies what program they're installing. The image must be a bitmap or JPG file that's 480 pixels wide and 320 pixels high. To set the image to display in this dialog box, first add the image to an appropriate folder in the File System editor, select the Splash dialog box in the User Interface editor, and then set the *Splash-Bitmap* property in the Properties window to the image file you just added to the File System. Figure 13-2 shows a splash screen of a setup project with the cover of this book used as the image.

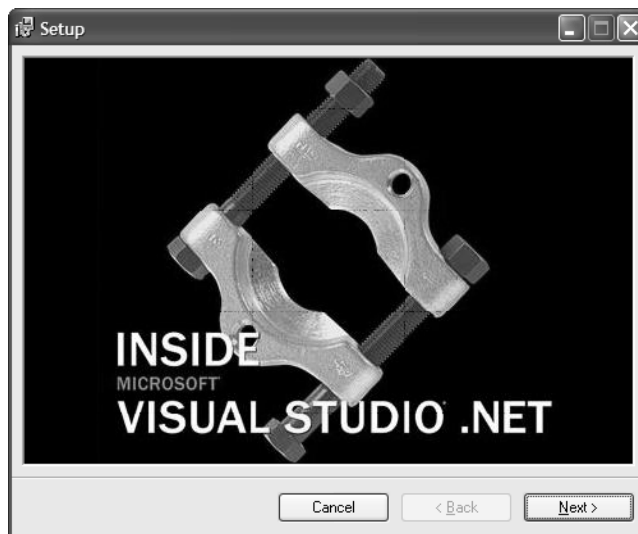


Figure 13-2 A splash screen for a setup project showing the Inside Microsoft Visual Studio .NET book cover

C13618747.fm Page 415 Friday, January 10, 2003 3:54 PM

Options Dialog Boxes

The options dialog boxes—RadioButtons (2 Buttons), RadioButtons (3 Buttons), RadioButtons (4 Buttons), Checkboxes (A), Checkboxes (B), and Checkboxes (C)—give you a way to offer users installation options. The RadioButtons dialog boxes display 2, 3, or 4 option buttons that the user can choose from, respectively; the Checkboxes dialog boxes display between 0 and 4 check boxes. (See Figure 13-3.)

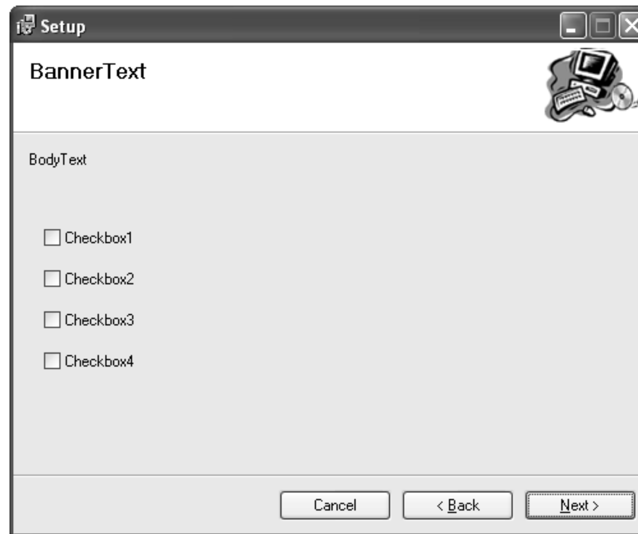


Figure 13-3 One of the Checkboxes dialog boxes

You can manipulate the settings for each option button and check box in these dialog boxes in the Properties window. You can set an option's default state (selected or unselected, checked or unchecked), installer property name, and label. The most interesting value is the installer property name, which is denoted in the Properties window as *ButtonProperty* for RadioButtons dialog boxes and *CheckBox*X*Property* (where *X* is a number from 1 to 4) for Checkboxes dialog boxes. You can use the value of these properties in other editors, such as the Registry editor, as key names, value names, or values, just as you would for the installer properties listed earlier in Table 13-1.

You can use the value of an option in a RadioButtons dialog box (shown in Figure 13-4) in the Registry editor in the same way you use a check box value. Only one option can be selected at a time, so only one property is available for each dialog box; the value of *ButtonXValue* is used when the installer populates the system registry. For example, the RadioButtons (2 Buttons) dialog box has two value properties, *Button1Value* and *Button2Value*, which are set to *1* and *2*,

C13618747.fm Page 416 Friday, January 10, 2003 3:54 PM

respectively. The property name of these buttons is *BUTTON2*, so in the Registry editor you can use the value *[BUTTON2]* for a registry key name, value name, or value. If the first radio button (*Button1*) is selected, the data placed into the registry is *1*; if the second button (*Button2*) is selected, the data is *2*.

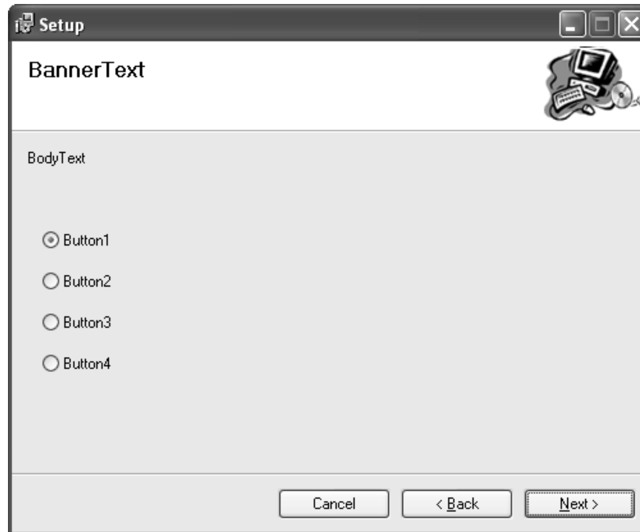


Figure 13-4 The RadioButtons (4 Buttons) dialog box

Later in this chapter, we'll discuss how you can use the options dialog boxes to conditionally install the registry keys and files you place in the Registry and File System editors.

Data Entry Dialog Boxes

You use the data entry dialog boxes—Textboxes (A), Textboxes (B), and Textboxes (C)—to ask the user for text data. Figure 13-5 shows one of these dialog boxes. Each data entry dialog box has four text boxes. For example, if you need to show only two of the text boxes, you can hide the other two text boxes by selecting the appropriate data entry dialog box in the User Interface editor and then, in the Properties window, setting the *Edit*X*Visible* property (where *X* is the edit box number) to *False*. Each text box in a dialog box has a name, and this name is listed in the Properties window using the *EditXProperty* property. You can use the value of this property, surrounded by square brackets, in the Registry Editor just as you use the other installer properties. For example, the first text box in the Textboxes (A) dialog box has the value *EDITA1* for the

C13618747.fm Page 417 Friday, January 10, 2003 3:54 PM

Edit1Property property, so you can enter the registry key name, value name, or the value *[EDITA1]* to represent what the user typed in the text box.

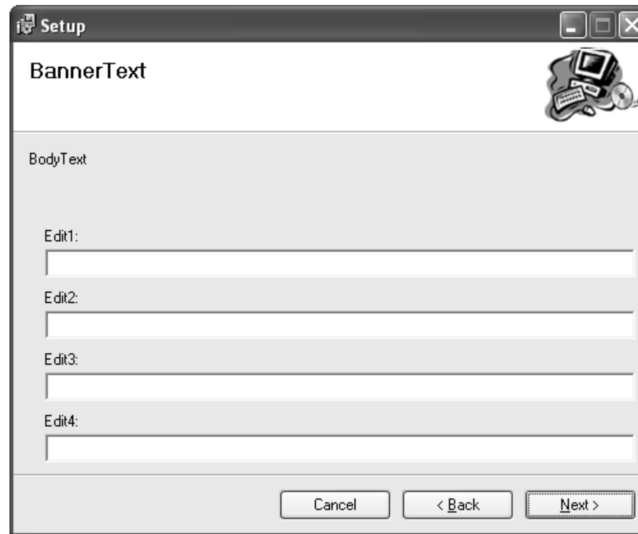


Figure 13-5 One of the TextBoxes dialog boxes

Customer Information Dialog Box

You use the Customer Information dialog box to gather information from users such as their name, the company they work for, and optionally a serial number for the program. To verify that a user has entered a correct serial number, a validation algorithm is performed on the serial number, with the algorithm being based on the value of the *SerialNumberTemplate* property in the Properties window. The value of the *SerialNumberTemplate* property creates the user interface for the serial number in the Customer Information dialog box and verifies that the serial number entered is valid. To define the serial number that the user can enter, you string together a special set of tokens to create a template. This template is surrounded by the less-than and greater-than symbols. Between these two characters, you can place any number of the characters #, %, ?, ^, and -. The # and % symbols are placeholders for digits, and ? and ^ are placeholders for alphanumeric characters, with ^ denoting an uppercase character. When a dash character is encountered within the template, a new text box is created in the dialog box. The dialog box determines whether the serial number entered is valid by adding all the numbers appearing in place of % in the template and dividing by seven. (The dialog box ignores all other characters in the template.) If the remainder is 0, the number entered by the user is considered valid and the user is allowed to continue installing the application;

C13618747.fm Page 418 Friday, January 10, 2003 3:54 PM

otherwise, an error message is shown and the user must either reenter the serial number to continue or exit the installation program. The dialog box shown in Figure 13-6 uses the default serial number template of <###-%%%%%%%>; the number entered is invalid because the sum of the numbers 4, 5, 6, 7, 8, 9, and 9 is 48, which is not evenly divisible by 7. Once the serial number has been validated, the value of the serial number entered is stored in the installer *ProductID* property.

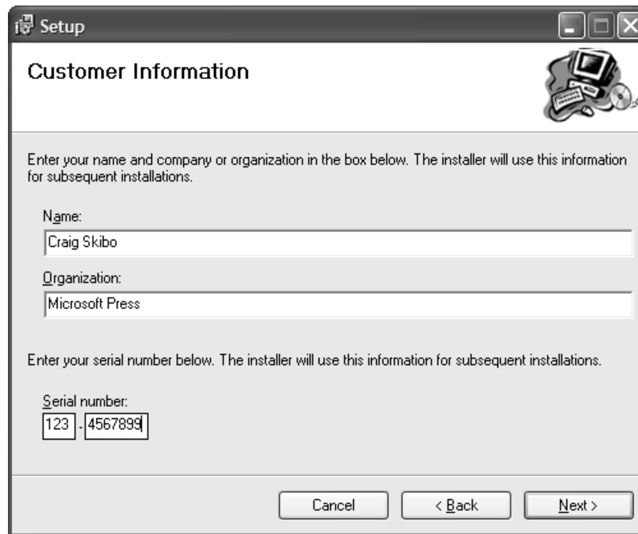


Figure 13-6 The Customer Information dialog box

License Agreement and Read Me Dialog Boxes

Just about every software program comes with some legal restrictions on how the program can and cannot be used. These restrictions, in the form of a license agreement, generally inform the user that the software cannot be illegally copied, cannot be reverse-engineered, and so forth. You can use the License Agreement dialog box to display license information to the user; unless the user selects an option to accept the license agreement, the user cannot continue installing the software. The options for accepting or not accepting the terms of the license agreement are shown in Figure 13-7. If the user selects the I Do Not Agree option, the Next button is disabled. Selecting the I Agree option is legally binding; a user who accepts the license agreement can continue installing the program.

To create the text to display in the License Agreement dialog box, you must create a rich text format (RTF) text file. You can use tools such as Microsoft Word or even WordPad, the better-than-Notepad text editor installed

C13618747.fm Page 419 Friday, January 10, 2003 3:54 PM

with Windows, to create an RTF file. After creating this file, you can add it to the File System editor and then set the text to appear in the License Agreement dialog box by first adding and then selecting the License Agreement dialog box in the User Interface editor and then in the Properties window browsing to the RTF file with the *LicenseFile* property.

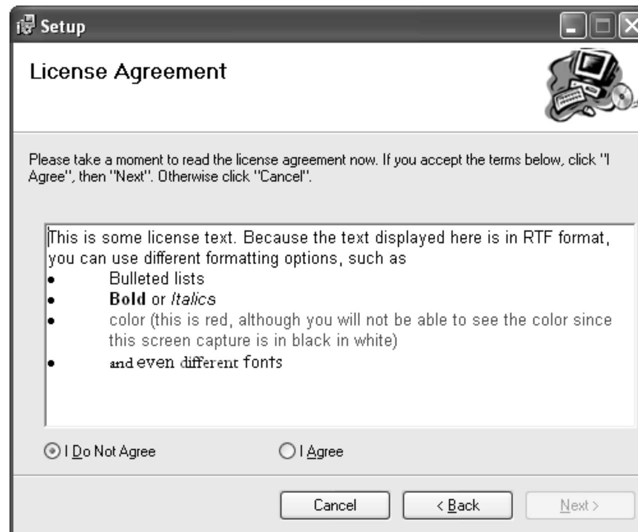


Figure 13-7 The License Agreement dialog box with richly formatted text

The Read Me dialog box looks and works much like the License Agreement dialog box, except it doesn't have I Do Not Agree and I Agree options. Text in the Read Me dialog box, like that in the License Agreement dialog box, is defined using an RTF file, so it can contain text that's formatted with colors, fonts, and styles.

Register User Dialog Box

Many software packages ask users for personal information such as name and e-mail address so when new versions or bug fixes are available, the software company can send them upgrade information. You can add the Register User dialog box, shown in Figure 13-8, to your setup program to gather this information. This dialog box doesn't contain entries for users to type their name, e-mail address, land-based address, or other data; instead, it contains a simple button that, when clicked, invokes any executable program that has been added to the File System editor of your setup project.

C13618747.fm Page 420 Friday, January 10, 2003 3:54 PM

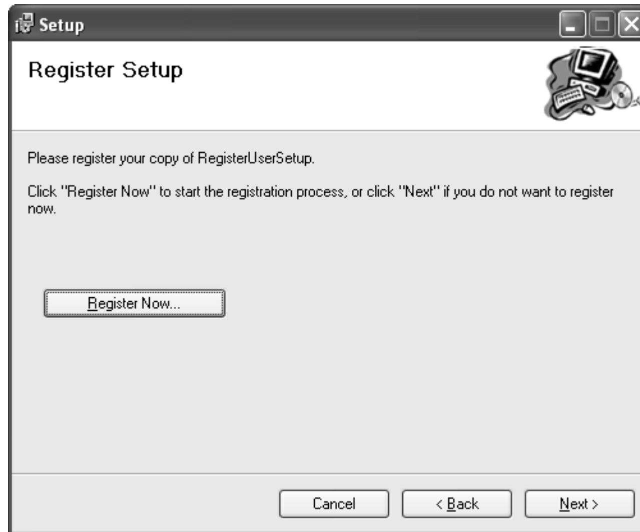


Figure 13-8 The Register User dialog box

To associate your registration program with the Register Now button, you must first create a registration program and add the primary output of this program to your setup project's File System editor. Next, select the Register User dialog box in the User Interface editor, and in Visual Studio .NET's Properties tool window, browse to the executable using the *Executable* property.

Creating the registration program isn't complicated—you can simply run the C# Windows Application Wizard and use the executable file—but you can find the source code for a registration project among the book's sample files. This project, RegisterUser, gathers the appropriate information from the user. Another sample project, called ProductRegistration, is a .NET Web application that you can install on a Web server. When the RegisterUser program is run and the user clicks the Register Online button, the information entered is packaged up into a HTTP request header and then posted to the ProductRegistration Web application. The Web application then unpacks this information from the request header and sends a message back to the RegisterUser project that can be displayed to the user.

To use the RegisterUser and ProductRegistration samples, install the ProductRegistration sample onto your Web server and add the output of the RegisterUser project to your setup program. Within the ProductRegistration code, open the code-behind file for the ProductRegistration.aspx file and find the TODO comment toward the end of this file. You should replace this comment with custom code to store the user's information for later use, such as within a

C13618747.fm Page 421 Friday, January 10, 2003 3:54 PM

database. You can also modify the text message returned to the user, customizing the message to suit your needs. The second step is to modify the RegisterUser Form1.cs file to point to the server containing the ProductRegistration Web application. To do this, search for the string *localhost* and change it to point to the server and virtual directory where the ProductRegistration Web application is installed.

Custom Actions

Windows Installer offers a lot of functionality to help you install your application, but at times you might need to run code during an installation to help get your program onto the user's computer. This helper code is called a *custom action*. We looked briefly at a custom action in Chapter 7 to help create and remove commands for Visual Studio .NET, but you can create a custom action to do anything you want. You can build three types of custom actions for a setup project: .NET Custom Actions, Script Custom Actions, and Win32 Custom Actions. The samples for this book include the CustomActions sample, which demonstrates creating and using each of these custom action types.

You add custom actions to the Custom Actions editor by first adding the primary output of the project implementing the custom action to the File System editor and then right-clicking on the appropriate installation action in the Custom Actions editor and browsing to the project output. Four installation actions are defined: Install, Commit, Rollback, and Uninstall. Which one you add your custom action to will depend on the work your custom action performs. An Install custom action is called when an .msi file is being installed. If an error occurs during installation, custom actions in the Rollback group are called. A Rollback custom action is run when an error occurs during installation; it can be used to repair a computer, removing data such as files or registry keys created within the Install action. If the installation completed successfully, custom actions in the Commit group are called, allowing you to complete setup on the computer. When a program is uninstalled, custom actions in the Uninstall group are called, giving your custom action the chance to clean up any data that might have been left behind by your program.

.NET Custom Actions

You can build custom actions by using the .NET Framework with an executable program, such as a Windows Forms application or a console application, or with a .NET class library that derives from a class found in the .NET base class libraries. Choosing which type of custom action to create is a tradeoff between how your user interacts with your custom action and how easy it is for you to develop and test your custom action.

If you want your custom action to display a user interface, the best option is to create a Windows Forms custom action. You simply add a Windows Forms application to your solution, add the output of this project first to your setup application's File System editor and then to the Custom Actions editor, and then select the appropriate installation stage in which the custom action should be run. When that stage is run, the custom action program is called, allowing the user to interact with the user interface the custom action displays. Creating a Windows Forms custom action is also a good choice for ease of developing and testing your custom action. Because a Windows Forms custom action is a program, you can run, test, and debug the custom action without needing to build and install the .msi file; this increases your productivity. Creating a custom action with a Windows Forms application is not the best choice if you don't intend to display a user interface because the form will block the install progress until dismissed, and you don't want to display a dialog box to the user that simply says "Click me to continue installing"—that's just poor style.

The second option is to create a console application custom action. This custom action type offers the benefits of a Windows Forms custom action in terms of the ease of testing and debugging, and it displays a user interface to the user. Depending on what your custom action does, the user interface can be either a blessing or a curse. Users don't like seeing console windows—they're not pretty to look at and are hard to use. If the custom action you're creating does its job quickly, the screen will flash with a console application, causing the user to question what the installer is doing to his computer. However, if you need to display text information such as the output of another console application, a console custom action is a good choice. If you've created a custom action using either a Windows Forms or console application using a language supported by the .NET Framework, you must change the *InstallerClass* property in the Properties window from *True* to *False* when the custom action is selected in the Custom Actions editor. If this property is *False*, Windows Installer knows it should invoke the custom action as a program. If this property is *True*, Windows Installer searches the program for a class with a specific attribute, which is used by the third type of .NET custom action—a .NET class library.

A .NET class library is a good choice if your custom action should run silently in the background without any user interaction. A custom action of this type is more complicated to test and debug because a class library isn't a free-standing executable that you can run without a hosting application. To create a class library custom action, you create a class library using your favorite .NET-enabled programming language, right-click on that project in Solution Explorer, choose Add | Add New Item, and then add an Installer Class item to the project. The item added is a class that derives from the class *System.Configura-*

C13618747.fm Page 423 Friday, January 10, 2003 3:54 PM

tion.Install.Installer and uses the attribute *RunInstaller*. When the .msi project is run and starts to run custom actions, it examines all classes within an assembly that implement the custom action; if it finds a class with the *RunInstaller* attribute, the class is instantiated and a proper method of the class is called. The class *System.Configuration.Install.Installer* defines four methods you can override that are called during certain actions of the install process: *Commit*, *Install*, *Rollback*, and *Uninstall*. You can add these to the class from the Class View window.

Script Custom Actions

Creating a custom action by using script code involves little more than creating a VBScript or JScript file, adding that file to the File System editor, and then adding it as a custom action in the Custom Actions editor. A script custom action is just a list of commands that the ActiveX Scripting engine loads and runs. When these script custom actions start running, one global variable of type *Session* named *Session* is created so you can find out information about the currently installing .msi file. Listing 13-1 shows a simple VBScript custom action, and Listing 13-2 shows its JScript equivalent. The custom action does little more than show a message box to the user containing these custom actions.

VBScriptCustomAction.vbs

```
'Can use the object Session to peek into the MSI
' file being installed. See the MSDN topic located at
' ms-help://MS.VSCC.2003/MS.MSDNQTR.2003FEB.1033/msi/vref_8xis.htm
' for information about how to use this object.
msgbox("VBScript Custom Action")
```

Listing 13-1 Source code for a VBScript custom action

JScriptCustomAction.js

```
//Can use the object Session to peek into the MSI
// file being installed. See the MSDN topic located at
// ms-help://MS.VSCC.2003/MS.MSDNQTR.2003FEB.1033/msi/vref_8xis.htm
// for information about how to use this object.
var wshShell = new ActiveXObject("WScript.Shell");
wshShell.Popup("JScript Custom Action");
```

Listing 13-2 Source code for a JScript custom action

You might choose to create a script custom action rather than another type of custom action for a couple reasons. First, you might have built up a library of scripts written using VBScript or JScript to help configure a computer. Rather than rewriting these into a .NET custom action, you can simply add the scripts to the setup project and run them. Another reason you might want to create a custom action using script is ease of development. Scripts are lines of text that

C13618747.fm Page 424 Friday, January 10, 2003 3:54 PM

are interpreted, which means you don't need to compile them. To create and test a script custom action, you simply open any text editor (even Notepad), write code, and then run that script code by double-clicking on the file in Windows Explorer.

Win32 Custom Actions

If the software you're trying to install is not a .NET application and the user doesn't have the .NET Framework installed on his computer, your choices are either script or Win32 custom actions. Script custom actions are easy to write but don't have full access to the Windows API and therefore might not be an option. The only remaining choice is to use a language such as Visual C++ to create a Win32 custom action. To create a Win32 custom action, you must first create a Visual C++ Win32 DLL and export a function that uses the *__stdcall* calling convention. This exported function must return a value of type *unsigned int*, which is a status code. If the custom action returns anything other than *ERROR_SUCCESS*, Windows Installer thinks it failed and rolls back the installation. The exported function takes as its only argument a value of type *MSIHANDLE* that can be used to query the setup project for information. To let Windows Installer know which exported function it should call within a DLL when a custom action is run, you must set the *EntryPoint* property in the Properties window when the custom action is selected in the Custom Actions editor. If you were to write a custom action like that shown in Listing 13-3, for example, you would enter **Install** for the *EntryPoint* property because that's the exported function for handling the custom action invocation.

Win32CustomAction.cpp

```
#include "stdafx.h"
#include <tchar.h>
#include <msi.h>
#include <Msiquery.h>

BOOL APIENTRY DllMain(HANDLE hModule, DWORD ul_reason_for_call,
                      LPVOID lpReserved)
{
    return TRUE;
}

__declspec(dllexport) unsigned int __stdcall Install(MSIHANDLE hInstall)
{
    MessageBox(NULL, "CustomAction", "VC++", MB_OK);
    return ERROR_SUCCESS;
}
```

Listing 13-3 Source code for a Win32 custom action

Lab: Debugging Custom Actions

Debugging a custom action when it's running inside an .msi file isn't as simple as it might seem. An .msi file doesn't run code—it's a collection of compressed files with data describing how those files should be installed. To debug a custom action, you must start installing the .msi file and attach the debugger at just the right time—after the code has started executing but before the code you want to debug has run. As you can guess, getting the timing just right can be nearly impossible. A trick I use to debug custom actions is to place a message box inside the custom action just before the code that I want to debug and then build and install the .msi file. When the message box appears, I know I can attach the debugger to that code and start debugging. Execution of the custom action stops while the message box is shown, so I know I'm connecting the debugger at the correct time. But what program do you attach the debugger to? It depends on which type of custom action you created. If the custom action is a .NET class library and you open the debugger's Processes dialog box, you'll see that three msiexec.exe processes are running that you can attach to. One of these processes will have the word *.NET* in the Type column of this dialog box. This is the process to attach to because it has the .NET Framework loaded and is executing your custom action.

If the custom action you created is a .NET Windows Forms application or a .NET console application, you'll see the programs listed in the Processes dialog box by their executable names; you can attach to those processes without attaching to the msiexec.exe process. To debug a Win32 DLL custom action, you must look in the Processes dialog box for an msiexec.exe process that has the same title you used for your message box in the Title column. Currently, there's no way to debug a script custom action; you have to use an alternative method of debugging, such as displaying a message box with information so you can trace through the code.

When you're done debugging, don't forget to remove your message boxes; otherwise, the user will see them when installing an .msi file.

Launch Conditions Editor

Sometimes the software you create will have special requirements for running. For instance, suppose you take advantage of the latest technology in Windows XP and therefore the user must run that operating system to run your program.

C13618747.fm Page 426 Friday, January 10, 2003 3:54 PM

Or suppose your program is an add-in and therefore cannot run without Visual Studio .NET installed on the computer. You could write a custom action that checks for these requirements during install time, but it's better to let the .msi file ensure that these requirements are met before the files are placed on the user's computer.

You can define the requirements in a setup project with a launch condition in the Launch Conditions editor. To add a launch condition, open the Launch Conditions editor, right-click on the Launch Conditions node, and choose Add Launch Condition. In the Properties window, you can type an expression in the *Condition* property that must evaluate to *true* for the .msi file to install. If this expression doesn't evaluate to *true*, the user cannot install your program and sees the error message contained in the *Message* property. If the *InstallURL* property for the condition is set to anything other than an empty string and the condition evaluates to *false*, the user has the option to go to a Web page to find more information about why installation failed. A condition expression must use a special syntax, or condition algebra, in order for Windows Installer to be able to evaluate the expression.

Condition Algebra

To define a condition, you must use a Visual Basic .NET–like syntax to define an expression. The variables you can use in an expression are the same installer properties listed earlier in Table 13-1 or those found in the various dialog boxes in the User Interface editor. However, when you define a condition, you don't need to include the brackets around the installer property names as you do in other editor windows.

Table 13-2 lists the operators you can use in an expression. These operators, when combined with string or integral constants (floating-point comparisons aren't supported) and installer property names, create the algebra you use to create a condition.

Table 13-2 Condition Algebra Operators

Operator	Description
Not	Logically negates the term.
And	*True* if the two terms evaluate to *True*, *False* otherwise.
Or	*True* if one of the two terms is *True*.
Xor	*True* if only one of the two terms is *True*.
Eqv	Logical equivalence operator. *True* if both terms are *True* or both are *False*.

C13618747.fm Page 427 Friday, January 10, 2003 3:54 PM

Table 13-2 Condition Algebra Operators *(continued)*

Operator	Description
Imp	Implication operator. *True* if the left term is *False* or the right term is *True*.
=	Equality operator. *True* if both terms are equal; otherwise evaluates to *False*.
<>	*True* if the two terms are not equivalent.
>	*True* if the left term is greater than the right term.
>=	*True* if the left term is greater than or equal to the right term.
<	*True* if the left term is less than the right term.
<=	*True* if the left term is less than or equal to the right term.
><	Bitwise *And* operator. *True* if any bits in the two terms match.
><	String comparison operator. *True* if the left string contains the right string.
<<	Bitwise comparison operator. *True* if the high 16 bits of the left integer term equal the right term integer.
<<	String comparison operator. *True* if the left string starts with the right string.
>>	Bitwise comparison operator. *True* if the low 16 bits of the left integer term equal the right term integer.
>>	String comparison operator. *True* if the left string ends with the right string.

Here are some examples of using these operators and installer Properties in the *Condition* property:

- ***Not Privileged*** *True* if the user isn't running under elevated privileges.
- ***(VersionNT = 501) And (ServicePackLevel = 1)*** *True* if the .msi file is being installed on Windows XP (version 501) and with Service Pack 1 installed.
- ***(VersionNT = 500) Or (VersionNT = 501)*** *True* if the .msi file is being installed on Windows 2000 (version 500) or Windows XP (version 501).
- ***(ScreenX >= 800) And (ScreenY >= 600)*** *True* if the user is running at a screen resolution of 800 × 600 or greater.

C13618747.fm Page 428 Friday, January 10, 2003 3:54 PM

- ***PhysicalMemory > 128*** *True* if the computer has more than 128 MB of memory installed.
- ***PhysicalMemory >= 128*** *True* if the computer has 128 MB or more of memory installed.
- ***"Hello World" >< "Hello"*** *True* if the string on the right is contained in the first string.
- ***"Hello World" << "Hello"*** *True* if the string on the left starts with the string on the right.
- ***"Hello World" >> "World"*** *True* if the string on the left ends with the string on the right.
- ***Intel > 4*** *True* if the computer's processor is an Intel Pentium or later. This rule is useful if your software is compiled to use only the Pentium (or compatible) processor instruction set.
- ***Intel = "5"*** *True* if the computer's processor is an Intel Pentium. This expression is similar to the previous one, except with the number 5 is surrounded by quotes, the greater-than operator cannot be used because you cannot evaluate a string that includes a numerical operator.
- ***BUTTON2 = 1*** *True* if you added the RadioButtons (2 Buttons) dialog box to the User Interface editor and the user has selected the first option button in that dialog box.

Defining Custom Installer Properties

You've seen the use of installer properties in the Registry, Launch Conditions, User Interface, and File System editors. However, these installer properties, defined by either Windows Installer or dialog boxes added to the setup project, might not always meet your needs. Using the Launch Conditions editor, you can create custom installer properties to use wherever installer properties are allowed. To create a custom installer property, right-click on the Search Target Machine node in the Launch Conditions editor and choose File Search or Registry Search.

A File Search installer property searches the computer for a file, and if the file is found, the property is set to the path of that file. To specify the file to search for, add a File Search launch condition and then set the folder in which to start the search and the file to search for (in the Properties window's *Folder* and *FileName* properties, respectively). If you know that the file to search for is

C13618747.fm Page 429 Friday, January 10, 2003 3:54 PM

somewhere in one of the subfolders of the folder specified, you can set the *Depth* property to the number of folder levels into the folder hierarchy that should be searched.

For example, suppose you need to set an installer property to the path of the file dte.olb, which contains the type library for the Visual Studio .NET object model. Because Visual Studio .NET installs this file in the Program Files folder, you set the *FolderName* property of a file search launch condition to *[ProgramFilesFolder]*, the *File* property to *dte.olb*, and the *Depth* property to *20* (an arbitrary value that will ensure that all the necessary folders are searched). If the installer property name of the file search launch condition is *FILEEXISTS1* (the default installer property name of the first file search launch condition created), you can use the installer property *FILEEXISTS1* in the File System editor, Registry editor, or any other place that installer properties can be used where the path to dte.olb is needed.

A Registry Search launch condition works much like a File Search launch condition, except it searches the system registry rather than the user's disk drive, and an installer property is set to a registry value rather than a file path. For example, suppose you want to verify that Visual Studio .NET 2003 is installed before you try to install an add-in. You can do this by using a Registry Search condition. First, create a Registry Search condition by right-clicking on the Search Target Machine node, choosing Add Registry Search, and typing a name for the condition. In the Properties window, type the registry key hive for Visual Studio .NET in the *RegKey* property, which is **SOFTWARE\Microsoft\VisualStudio\7.1**, and type **InstallDir** as the Value property. The *InstallDir* registry value holds the location on disk where Visual Studio .NET has been installed. The other values can be left as their defaults, but you should note the value of the *Property* property. This is the installer property you'll use for creating the condition. (By default, this value is *REGISTRYVALUE1* for the first registry search condition.) Next, create a condition by right-clicking on the Launch Conditions node in the Launch Conditions editor and choosing Add Launch Condition. Open the Properties window, type **REGISTRYVALUE1 <> ""** for the *Condition* property, and type any message you want in the *Message* and *InstallURL* properties. When the .msi file for the Add-in is run, it creates an installer property called *REGISTRYVALUE1* that's set to the installation location of Visual Studio .NET. If this installer property is anything other than an empty string, the expression is *True* and setup continues; otherwise, a message is shown to the user with the value you entered in the *Message* property.

Installing an Assembly in the PublicAssemblies Folder

As we've said a couple times in this book, if you build an assembly that you want to call from a macro, you must put the assembly into a specific folder so you can add a reference to that assembly in the Macros editor. If you use the default installation location of Visual Studio .NET, which is C:\Program Files\Microsoft Visual Studio .NET 2003\Common7\IDE\PublicAssemblies, you cannot assume it's correct because the user might have selected a different installation location for Visual Studio .NET. You can use the custom *Registry Search* installer property we just described to verify that Visual Studio .NET is installed, and then install the assembly in the correct location so it can be used by a macro.

To place the assembly in the correct place, open the File System editor, right-click on the File System On Target Machine node, and choose Add Special Folder | Custom Folder. In the Properties window for this custom folder, type the name of the custom *Registry Search* installer property surrounded by square brackets (in this case, *[REGISTRYVALUE1]*), and then type **\PublicAssemblies** in the *DefaultLocation* property. Then add the primary output of the assembly to the custom folder. When the .msi file is installed, it places the assembly in the correct location so it can be referenced in the Macros editor.

Conditions

If you looked closely at the Properties window while working with the File System, Registry, Custom Actions, or File Types editors, you might have noticed the *Condition* property. This property controls whether a project output, registry key, or file type is installed on the computer or if a custom action is run. You can use the same condition algebra you use to create a launch condition to set the *Condition* property. For example, suppose you want to give the user the option of installing the source code for an add-in that you created. To do this, you run the Add-in Wizard to completion, and after the source files have been generated, you open the User Interface editor for the setup project. Insert a RadioButtons (2 Buttons) dialog box into the User Interface editor, open the Properties window, and set the *Button1Label* property to **Yes, install source code** and set the *Button2Label* property to **No, do not install source code**. Make note of the installer property name of this dialog box, *BUTTON2*, and the values of the buttons in the dialog box, *1* and *2*.

Next, open the File System editor for the setup project, right-click on the Application Folder node, choose Add | Folder, and then enter **source**. This creates the folder to hold the source files for the add-in. To add the source files output, right-click on the source folder, choose Add | Project Output, and select the Source Files output. If you build and install the .msi file for the add-in after making these changes, the source files are always installed because the condition on the source files output isn't connected to the option the user selects in the RadioButtons (2 Buttons) dialog box. To connect the dialog box to the source files folder you created, select the source folder in the File System editor, and in the Properties window enter the expression **BUTTON2 = 1** in the *Condition* property. Now if you install the .msi file and if the user selects the first option button, the sources are installed in a folder called sources. But if the user selects the second option button—the one that corresponds to the value *2*—the expression *BUTTON2 = 1* evaluates to *False* and the source folder isn't created. Because the source file output is contained in the source folder, that project output is also not installed. You could place the condition *BUTTON2 = 1* on the source file project output so that if the user selects the second option button, the source files are not installed, but the source folder will have been created, leaving an empty folder on the user's computer. The samples for this book contain the setup project for the SourceFilesAddin sample, which demonstrates how you can optionally install the source code to an add-in project.

Merge Modules

When creating software for the Windows operating system, developers commonly place code into DLLs so it can be shared among multiple applications. To distribute software to your customers, you could install the DLLs you create with a .msi setup project, but this would be a less-than-ideal way to install the code. Suppose you've built a .NET user control library that you want to sell; your customers can use this library to build their own applications. How can customers redistribute this library to their own customers? You could provide detailed instructions that explain to your users how to include the component in their own .msi setup. However, this could be problematic because if they don't install your component properly, other software that uses your component might stop functioning. Alternatively, you could create a setup project that installs and uninstalls your library, but that's not a good option because your users would have to give their customers two .msi files and make sure they were installed in the correct order.

To make installing your library easy and reduce the risk of a user incorrectly installing and uninstalling a component, you can store your library in a

C13618747.fm Page 432 Friday, January 10, 2003 3:54 PM

merge module (.msm file). Merge modules are like the DLLs of a setup project. You can use a DLL to store code shared among different applications; a merge module contains installation logic shared among many .msi files. Merge module projects are similar to setup projects, except you cannot use the User Interface and Launch Conditions editors. One other difference between a setup project and a merge module project is that with a merge module project, the File System editor adds the folders Web Custom Folder and Module Retargetable Folder. If you place files in these folders, the user consuming your merge module can choose which location on disk to place the files. The user can change the installation path of the merge module contents by selecting the merge module in Solution Explorer and modifying the *ModuleRetargetableFolder* property in the Properties window. Consuming a merge module in a setup project is easy; you select the setup project in Solution Explorer and choose Project | Add | Merge Module.

Setup for .NET Programs

Suppose you're taking the setup-building capabilities of Visual Studio .NET out for a spin, trying out the various features. You've built a .NET application, added the project output to a setup project, built the solution, and tested the .msi file by installing the project. Everything has installed perfectly, and you were able to run your application. Being the good developer you are, and because you want to be sure your program works in all scenarios, you try installing the same .msi file on a clean computer—a computer with nothing more than the base operating system installed. When you run the setup project on the clean computer, you're presented with the dialog box shown in Figure 13-9. What went wrong, and why did it work fine on the development computer but not the clean computer? Visual Studio .NET cannot run without the .NET Framework installed, and the clean computer, with only the operating system installed, doesn't have the .NET Framework.

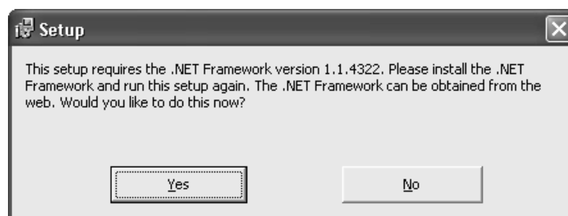


Figure 13-9 The error message you see when you try to install a .NET program on a computer that doesn't have the .NET Framework installed

C13618747.fm Page 433 Friday, January 10, 2003 3:54 PM

When you add output from a C# or Visual Basic .NET project to the File System editor, Visual Studio .NET automatically adds a condition to the Launch Conditions editor that verifies that the .NET Framework is installed. Because the clean computer doesn't have the .NET Framework installed, the launch condition was not satisfied and an error was displayed to the user to that effect.

To run an .msi file that contains components that use the .NET Framework, you have two options. The option you choose will depend on how you plan on distributing your software. The first option is to let the user click the Yes button in the dialog box shown in Figure 13-9 and then install the .NET Framework manually. This option is best if you're distributing your software on the Internet or using a limited-size medium such as a floppy disk that doesn't have enough room to hold the .NET Framework redistributable files.

If you're shipping your program on a large media format such as a CD, another option is better. You store the .NET Framework redistributable files on the CD, and use a bootstrapping program to install the .NET Framework and then your .msi file. A bootstrap program is a small bit of code that takes care of getting the installation up and running. The bootstrap program makes sure the .NET Framework is installed and then starts installing the .msi file. To install a program, you should run the bootstrap program but not run the .msi file. By default, when you build a setup project, a bootstrap program named Setup.exe is generated and placed in the output folder for your setup project. This bootstrap program only makes sure that the Windows Installer program is installed, and not the .NET Framework. You don't need to redistribute this bootstrap program if you're trying to install the .NET Framework because the setup program for the .NET Framework installs the .msi installer if it isn't on the user's computer, and you can turn off generating this bootstrap file in the setup project Properties dialog box.

When you first add a .NET component to a setup project, a dependency to the merge module dotnetfxredist_x86.msm is added to your project and is marked to be excluded. You might assume that you can include this merge module in your setup project (instead of exclude it) to set up the .NET Framework if needed, but this is not what this merge module does. This merge module contains assemblies that are part of the .NET Framework—such as System.XML, System.Web, and so forth—and should be on the user's computer if the use has the .NET Framework installed. This merge module doesn't contain files that make up the common language runtime (CLR), so if you were to include this merge module in your setup project, you'd be including many assemblies that the user should already have installed but not everything the user needs to run a .NET program.

To make installing the .NET Framework with your program easier, you can use the Bootstrapper sample, which is included with the samples for this book. This sample, written using Visual C++, performs a couple of steps when it first runs. First, it checks to make sure another instance of the bootstrapping program isn't running because if one is already running, errors can arise if the other instance is started. To ensure that another instance is not running, the bootstrap program creates a mutex; mutexes are shared across process boundaries, so an error is generated if the mutex has already been created by another instance of the bootstrap program. This error condition signals that another instance is running; a message is displayed to the user and then the bootstrap program exits. The second step the bootstrap program performs is to read a file called Setup.ini, which needs to be located in the same folder as the bootstrap executable. This file describes to setup where it can find, among other things, the .NET Framework redistributable file. An example Setup.ini file is shown here:

```
[Setup]
InstallName=Setup
FrameworkVersionRequired=v1.1
UseLocaleForFindingRedist=1
FrameworkRedistName=DotNetfx
FrameworkInstallPath=FrameworkRedist
MSIFilePath=Setup.msi
```

All the values in this INI file are optional; if a particular key and value aren't found, the value as shown in this example is used. The meanings of these values are:

- ***InstallName*** The name of the product being installed. The value defaults to *Setup* if a name is not supplied. This string is used to display the name of your program in the user interface of the bootstrap program.
- ***FrameworkVersionRequired*** The version of the .NET Framework in use by the program being installed. If a currently installed .NET Framework with the version that matches this string is found, installation of the .NET Framework is skipped. This value can be *v1.0* (the letter *v* must precede the version number, and the trailing *0* is required) or *v1.1*. *v1.0* is the version of the .NET Framework installed with Visual Studio .NET 2002, and *v1.1* is the version of the .NET Framework installed with Visual Studio .NET 2003.
- ***FrameworkRedistName*** The name of the executable file (without the .exe filename extension) that holds the .NET Framework

C13618747.fm Page 435 Friday, January 10, 2003 3:54 PM

redistributable file. The name of this executable is always DotNetfx, so in most situations you don't need to specify this value.

- ***UseLocaleForFindingRedist*** If this value is *1*, the bootstrap program attempts to find and install a localized version of the .NET Framework redistributable from the CD. Up to this point, the .NET Framework has been localized into nine languages: English, French, German, Italian, Japanese, Spanish, Chinese Traditional, Chinese Simplified, and Korean. If a localized version needs to be installed, the bootstrap program retrieves the language used by the operating system and uses this language, or *locale*, when searching for the redistributable. If this value is *0*, the bootstrap program doesn't use the operating system language when searching for the redistributable file. If your program is not localized into different languages, you should distribute the localized version of the .NET Framework that your user customer would use and set this value to *0*.
- ***FrameworkInstallPath*** The path relative to the bootstrap program where the .NET Framework redistributable file can be found. Suppose you have the bootstrap code in the folder D:\MyProgramSetup and the *FrameworkInstallPath* value is set to its default value, *FrameworkRedist*. The bootstrap program will look for the .NET Framework redistributable file in the path C:\MyProgramSetup\FrameworkRedist\DotNetfx.exe. If the value of *UseLocaleForFindingRedist* is set to *1*, the language identifier is inserted into this path between the value for *FrameworkInstallPath* and the *FrameworkRedistName* value. Table 13-3 shows the language identifiers for the various languages used; if the language is English, for example, the redistributable is searched for at D:\MyProgramSetup\FrameworkRedist\9\DotNetfx.exe. If the redistributable is not found in the path with the language identifier or if a language being used by the operating system is not supported, the path without the language identifier is searched. If all the search options have been exhausted and the redistributable hasn't been found, an error is given and setup exits.
- ***MSIFilePath*** The path relative to the bootstrap program where the setup .msi file can be found. If the bootstrap program is in the folder D:\MyProgramSetup and the default value of *Setup.msi* is used for this key, the path searched is D:\MyProgramSetup\Setup.msi.

C13618747.fm Page 436 Friday, January 10, 2003 3:54 PM

Table 13-3 Languages and Their Identifiers

Language	Language Identifier
English	*9*
French	*12*
German	*7*
Italian	*16*
Japanese	*17*
Spanish	*10*
Chinese	If the operating system is using Chinese Traditional, the language identifier is 1228. Otherwise, Chinese Simplified is used, which is language ID 2052.
Korean	*18*

> **Note** Installing the proper language version of the .NET Framework is important because although it might seem that only the program being installed will be affected, your users have to live with the language of the .NET Framework you install unless they uninstall and then install a new version of the language they want to use.

Creating a Setup CD

Today, most computers have a CD burner installed, and you can find blank CDs at your local discount store for well under 50 cents each. The availability of CD burners, cheap media, and the setup development tools available with Visual Studio .NET make distributing your software programs easy. To create a CD with your .msi file on it, you must combine the bootstrap program, the .msi file to install your program, and a few other files to make installation as seamless as possible for your users.

> **Note** If you intend to place the .NET Framework redistributable on the CD for your program or make it available for download from a non-Microsoft Web site, you must read and agree to the Microsoft .NET Framework SDK end user license agreement (EULA) before redistributing the .NET Framework.

C13618747.fm Page 437 Friday, January 10, 2003 3:54 PM

The Windows operating system simplifies installing software on a CD with AutoPlay. When a CD is inserted into the computer's CD drive, Windows examines the root folder of the CD, and if it finds a file called autorun.inf, it reads, parses, and then does the work as described in that file. Here's an example autorun.inf file that you can place in the root folder of a CD to automatically start running the bootstrap program when the CD is inserted into the drive:

```
[autorun]
open=setup.exe
icon=setup.ico
label=My Setup
shell\launch\command=Setup.exe
shell\launch=&Install this program
```

The meanings of each entry in this file are:

- ***open*** When the CD is inserted into the CD drive, this is the path to a file (without the drive letter) that will be run. If you're using the bootstrap program to install the .NET Framework and your program, you should give the path to the bootstrap program; otherwise, you can specify the relative path to the .msi file.
- ***icon*** The path to an icon on the disk (without the drive letter) that's displayed in Windows Explorer for the CD. The path can point to an .ico file or to an executable or DLL file. If the path is to an executable or DLL file, the path should be followed by a comma and then the zero-based index of an icon within the resources of that file.
- ***label*** The text to display as a label next to the CD drive in Windows Explorer.
- ***shell*\verbname** If the user right-clicks on the CD drive in Windows Explorer, the text following this entry is shown on the shortcut menu. The text *verbname* is arbitrary and can be any string, as long as it contains only alphanumeric characters. The text following this key name can contain any character, and if the ampersand character is used, the accelerator key follows; use a double ampersand if you want the string to contain the ampersand character.
- ***shell*\verbname*command*** If the user right-clicks on the CD drive in Windows Explorer and chooses the command specified in the *shell*\verbname line, the program pointed to by the path specified is run. The *verbname* string here is also arbitrary, but it

C13618747.fm Page 438 Friday, January 10, 2003 3:54 PM

should match the name used in the previous line. The autorun.inf file can contain any number of these *shell*\verbname pairs (including 0 entries), with each item appearing on the CD drive shortcut menu, but each pair of items should use a matching but unique pair.

Once you create your autorun.inf file, you must gather all the files that will be placed on the CD; this requires you to download the .NET redistributable files from Microsoft's Web site. Because the redistributable file is over 20 MB in size, this file (or files, if you intend to offer your users localized versions of the redistributable files) is not included with the samples for this book. Figure 13-10 shows the layout of the CD with the typical components necessary to install your program. This layout (without the .NET Framework redistributable files) can be found among the samples for this book in the folder SetupCD. To build a setup CD, you simply copy your .msi file into this folder, burn its contents to a writable CD, and you're done! You've just created a professional setup for your software.

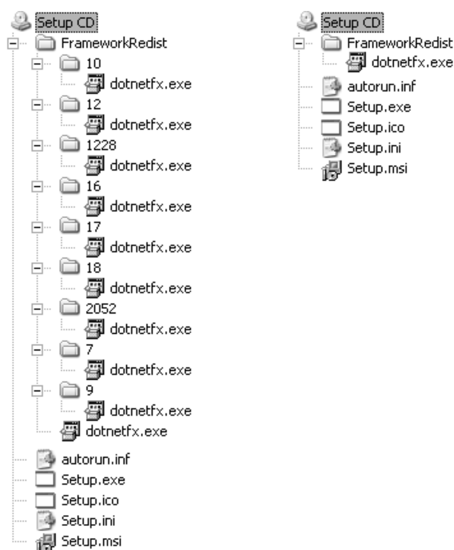


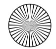
Figure 13-10 The directory structure of a setup CD if you offer localized versions of the .NET Framework (left) and a nonlocalized setup (right)

Setting Up the Book's Samples

If you downloaded and installed the sample files that accompany this book, you ran an .msi file that was built with the setup tools in Visual Studio .NET. This setup .msi file was built using many of the concepts described in this chapter, including installer properties, custom actions, the Registry editor, Web setup projects, and more. The source code for the setup project and the custom actions used can be found in the folder InsideVSNetSetup. For details about how to rebuild the Inside Visual Studio .NET 2003 samples .msi file, consult the Readme.htm file in the InsideVSNetSetup folder.

Looking Ahead

All users, regardless of their skill level, sometimes need help completing a task. Visual Studio .NET provides a full-featured help system to display MSDN contents. In the next chapter, we'll look at how you can use that system and how you can customize it to include your own help topics.



C13618747.fm Page 440 Friday, January 10, 2003 3:54 PM