



InstallShield Press

InstallShield software and
more on CD-ROM



**Topics Covered
Include:**

- Use of the many task-oriented wizards in InstallShield Developer
- In-depth discussion of how the Windows Installer works
- The architecture of InstallShield Developer
- Ins and outs of Windows installer components
- And much more

Getting Started with
**InstallShield
Developer**
and **Windows
Installer** Setups

BOB BAKER

“This book is a must-have resource for any developer seeking in-depth knowledge of the Windows Installer technology.”

www.installshield.com

- **Mark Lathrop**
Developer Support Lead
Microsoft Corp

InstallShield

**Getting Started with
InstallShield Developer and
Windows Installer Setups**

Bob Baker

PUBLISHED BY
InstallShield Press
A Division of InstallShield Software Corporation
900 N. National Parkway
Suite 125
Schaumburg, IL 60173

Copyright © 2002 by InstallShield Software Corporation

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

ISBN: 0-9715708-0-9
LCCN: 2002102340

InstallShield Press books are available through InstallShield Software Corporation. Information about purchasing InstallShield Press books can be obtained at www.installshield.com, by contacting Sales: TEL (800) 809-5659 or FAX (847) 619-0788.

The author and publisher have made every effort to verify that the material in this book is accurate. However, the information in this book is made available as is without any warranty of any kind, expressed or implied. The sample projects and source code included on the CD-ROM at the back of the book have a limited warranty, which is discussed in the End-User License Agreement accompanying the CD-ROM. The author, InstallShield Press, licensors and related parties will not be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Credits:

Project Editor: Lori Erickson

Technical Editors: Art Middlekauff, David Thornley, Richard Aquino, Marwan Tabet, Kent Foyer, Michael Marino, Martin Markevics, Gomathi Rajesh, Mingbiao Fei

Graphics Artist: Melvin Grefalda

*Dedicated to my children:
Bobby, Robin, Rebecca, Stefanie, and Shireen.
The future is yours.*

Contents

INTRODUCTION	XI
ACKNOWLEDGEMENTS	XVII
PART I - THE FUNDAMENTALS.....	1
Chapter 1 - Installation Development.....	3
Who Is Interested in Software Installation?.....	4
The End User.....	4
The LAN Administrator.....	5
The Setup Developer.....	5
What Operations Constitute a Typical Installation?.....	6
Checking the System.....	8
Checking for Required Applications	8
Checking Available Disk Space	9
The User Interface.....	10
Copying Files	11
Registering the Application	12
Making the Application Available.....	12
Maintaining The Application.....	13
What Are the Various Types of Installations?.....	13
The Fresh Install.....	14
The Maintenance Install.....	14
The Advertised Install.....	14
The Administrative Install	15
The Upgrade Install.....	15
The Patch Install.....	16
The Transformed Install.....	17
The Nested Install	17
Installation Development Technologies and Tools	18
Script-Based Installation Programs	18
Windows Installer-Based Installation Programs	23
InstallShield Developer.....	26
Conclusion.....	28
Chapter 2 - Introducing InstallShield Developer	29
A Quick Tour	30
The Opening Screen.....	30
The InstallShield Today Page	32
Help View and Best Practices View	37
The Menus.....	37
The Toolbar	42
The Sample Application.....	46
Creating a Standard Project Using the Project Wizard.....	47
Creating a Basic MSI Project Using the Project Wizard.....	73
Linking to Source Files at Build-Time	78

Conclusion.....	81
Chapter 3 - Windows Installer Basics.....	83
The Design Focus.....	84
The Windows Installer Package.....	88
What is a COM Structured Storage File?	90
The Summary Information Stream	94
The Windows Installer Database.....	94
Compressed and Uncompressed Source Files.....	96
The Windows Installer SDK.....	98
Dissecting the Developer Art Installation Package	99
The Developer Art Summary Information Stream.....	100
The Database Tables.....	105
The File Copy Tables.....	119
The Registry Entry Tables.....	122
The Installation Procedure Tables.....	124
The User Interface Tables.....	128
The Desktop Integration Tables.....	133
The Installation Validation Table.....	136
How Does the Windows Installer Perform an Installation?	138
Running the Windows Installer Engine from the Command Line.....	138
Running the Windows Installer Engine Programmatically.....	141
The Operations of the INSTALL Top-Level Action.....	142
Extending the Windows Installer Functionality.....	154
Custom Action Categories.....	155
The Types of Custom Actions.....	157
The Other Types of Windows Installer Packages.....	159
Merge Modules.....	159
Transforms.....	160
Patch Packages.....	161
Conclusion.....	161
Chapter 4 - The InstallShield Developer Run-Time Architecture.....	163
Fresh Install Run-Time Architecture.....	164
Fresh Install Using a Standard Project.....	164
Fresh Install Using a Basic MSI Project.....	184
Maintenance Install Run-Time Architecture.....	189
Maintenance Install Using a Standard Project.....	190
Maintenance Install Using a Basic MSI Project.....	195
Run-Time Architecture for Other Install Modes.....	196
Administrative Installations.....	197
Application Advertisement.....	198
Localized Installations.....	201
Run-Time Handling of InstallScript.....	207
Installing the InstallScript Engine.....	207
The Program Block and Event Handlers.....	208
InstallScript Custom Actions.....	211
Conclusion.....	215
Chapter 5 - Creating Projects in the IDE.....	217
Creating a Standard Project in the IDE.....	218
Organize Your Setup (Step 1).....	220
Specify Application Data (Step 2).....	243
Configure the Target System (Step 3).....	251
Advanced Views.....	257

Prepare for Distribution (Step 7).....	278
Creating a Basic MSI Project in the IDE.....	293
Conclusion.....	294

PART II - THE INSTALLSCRIPT LANGUAGE.....295

Chapter 6 - Variables and Data Types.....	297
Setting up the Programming Environment	298
Variables.....	302
Data Types.....	306
The Built-In Data Types	306
User-Defined Data Types	324
Conclusion.....	331

Chapter 7 - Expressions and Statements	333
Expressions.....	334
Arithmetic Expressions.....	334
Relational and Logical Expressions.....	339
String Expressions.....	347
Bitwise Expressions	351
Expressions Using the SizeOf and Resize Operators	359
Statements	361
Selection Statements	361
Iteration Statements.....	368
Jump Statements	376
Conclusion.....	377

Chapter 8 - Functions	379
Function Basics	380
The Built-In Functions.....	381
Built-In Function Prototypes and Definitions	381
Built-in Function Categories.....	383
The Function Wizard	386
Event Handler Functions.....	388
User-Defined Functions	400
Entry Point User-Defined Functions	400
Generic User-Defined Functions	401
Calling Functions in a DLL	434
User-Defined DLL Functions	434
Calling Windows APIs	441
Passing an Array to a DLL Function	445
Passing Strings to Functions.....	448
Conclusion.....	449

Chapter 9 - Exception Handling and COM	451
Exception Handling Basics	452
The try...catch...endcatch Statement	452
The Err Object.....	453
Exception Handling Hierarchy	455
InstallScript Engine Exceptions.....	459
Creating COM Objects.....	463
The Windows Installer Automation Interface	464
The Windows Installer Object Model.....	464
The Installer Object.....	466

The StringList Object.....	475
The Record Object.....	477
A Script Example.....	480
The Scripting Run-Time Objects.....	483
The FileSystemObject Object.....	484
The Dictionary Object.....	523
A Drives Collection Example.....	529
The Windows Script Host Objects.....	533
The Creatable Objects.....	534
The WshNetwork Object.....	538
The WshShell Object.....	542
More Objects.....	549
Conclusion.....	550

PART III - GETTING DOWN TO BUSINESS 551

Chapter 10 - Common Installation Tasks 553

Creating File Associations.....	554
The "Certified for Windows" Logo Requirements for File Associations.....	554
Creating a File Association for the Developer Art Application.....	555
Adding a MIME Type to the File Association.....	564
Defining Registry Entries.....	567
The Registry and RemoveRegistry Tables.....	568
Working with the Registry Table.....	575
Adding Registry Entries to the Developer Art Project.....	581
Removing Registry Entries During an Installation.....	584
Handling Environment Variables.....	587
Environment Variable Overview.....	588
The Environment Table Schema.....	589
Working in the Environment Variables View.....	593
Per-Machine vs. Per-User Installations and Environment Variables.....	598
Accessing Environment Variables During an Installation.....	601
Creating, Modifying, and Reading Initialization Files.....	605
The IniFile and RemoveIniFile Tables.....	606
Working in the INI File Changes View.....	609
Reading Initialization Files During an Installation.....	615
Searching for Files, Folders, and Registry Entries.....	618
How the Basic Search Mechanism Works.....	619
Basic Searching Examples.....	628
Checking for Compliance.....	640
Using Event Handlers for Searching.....	640
Miscellaneous Operations.....	648
Specifying Launch Conditions.....	648
Creating Empty Folders.....	650
Conclusion.....	652

Chapter 11 - InstallScript Custom Actions 655

Creating the Project.....	656
Preventing the User Interface Dialogs From Running.....	656
Preventing Project Registration.....	658
Defining a Feature and Component.....	659
Building the Project.....	660
Creating an InstallScript Custom Action.....	661
Creating the InstallScript Function.....	662
Using the Custom Action Wizard.....	667

Using a Custom Action	676
Testing the Custom Action	678
An Alternate Way to Create a Custom Action	679
Getting and Setting Properties	680
Example of Retrieving a Property Value	682
Example of Setting a Property Value	684
Accessing Database Tables	686
Example of Reading Values in a Table	688
Streaming Out Binary Data	693
The OnBegin and OnEnd Event Handlers	698
Using the MsiDoAction Function	700
Conclusion	702
Chapter 12 - User Interface Basics	703
The Basics of Windows Dialogs	704
Defining a Dialog	704
The Dialog Controls	706
Creating Dialog Functionality	709
Standard Projects vs. Basic MSI Projects	713
The Dialogs View in Standard Projects	714
Compiling Resource Files for a Standard Project	719
The Dialogs View in Basic MSI Projects	721
Generating a User Interface for a Standard Project	724
Understanding the Default User Interface	724
Modifying the Default User Interface	731
Creating a Custom Dialog Box	738
Generating a User Interface for a Basic MSI Project	774
Implementing the InstallNTService Dialog	774
Adding Serial Number Input to the CustomerInformation Dialog	786
Experimenting with Subscription	792
Conclusion	795
Chapter 13 - Introducing Components	797
Components and the Windows Installer	798
Some Definitions	798
Keeping Track of Components	800
The Component's Composition	803
Rules for Creating Components	806
The Component Creation Tools	811
Dynamic File Linking	812
The Component Wizard	825
Scanning	830
Special Considerations	840
Interfacing with Legacy Applications	840
Windows File Protection	843
DLL Redirection	845
Transitive Components	846
Qualified Components	847
Companion Files	848
Conclusion	850
Chapter 14 - Creating Special Components	851
COM Components	852
Win32 DLLs vs. COM DLLs	852
Why Self-Registration is Not Recommended	862

Recreating the ArtWork Component	866
NT Service Components	874
The NT Service Environment	874
Types of NT Services	877
Inside the Service Process	878
The Service Control Manager Database	880
Installing, Controlling, and Removing an NT Service	881
Using the Component Wizard to Install and Control an NT Service	882
Interactive Services	901
Installing an NT Service to a User Account	903
Font Components	906
ODBC Components	910
Overview of ODBC	910
Creating and Installing ODBC Components	915
Changes Made to the Operating System for ODBC	919
Merge Modules	921
Creating a Merge Module	922
Using a Merge Module	925
Conclusion	928
APPENDICES.....	929
Appendix A - The CD-ROM	931
Index	933

Introduction

InstallShield Developer is the most powerful installation-development tool ever released by InstallShield Software Corporation. This book provides all the essential information needed to start using this powerful product. *Getting Started with InstallShield Developer and Windows Installer Setups* is focused on the person new to the InstallShield Developer product and/or new to installation development. Some of the subject matter, however, will be of interest to experienced setup developers as well. This book contains a complete explanation of the two types of projects that can be created using InstallShield Developer.

Every attempt has been made to bring the content of this book into sync with version 7.03 of InstallShield Developer. However, many chapters were completed prior to the release of this product so there may be some areas that do not exactly match what you see in InstallShield Developer. The differences should be minor and will not cause any lessening of the value of the material contained herein.

About the book

This book is divided into three parts. The first part introduces both types of projects and includes chapters on the Windows Installer technology as well as the run-time

architecture of InstallShield Developer. It is explained how InstallShield Developer exploits the Windows Installer technology to create robust installation programs. The second part of the book provides complete coverage of the InstallScript programming language. The last part of the book goes into detail with regard to many of the basic techniques that can be used to create an installation package.

The following paragraphs provide a short description of the contents of the book.

Part I: The Fundamentals

CHAPTER 1: INSTALLATION DEVELOPMENT

This chapter introduces installation as a technology and discusses the three different viewpoints that the end user, the LAN administrator, and the end user have with regard to this technology. The typical types of installations and the types of operations that these installations carry out are reviewed. Finally there is a short history of the technologies and tools that have been used to create installations.

CHAPTER 2: INTRODUCING INSTALLSHIELD DEVELOPER

This chapter is an introduction to InstallShield Developer. By way of introduction the Project Wizard is used to create both a Standard project and a Basic MSI project. These projects install the sample application that is used through out the book. Through the use of the Project Wizard a number of important issues are raised and discussed.

CHAPTER 3: WINDOWS INSTALLER BASICS

Because both types of installation packages created by InstallShield Developer exploit the Windows Installer technology from Microsoft, this chapter provides a detailed overview of this technology. After discussing the basics of the Windows Installer technology a discussion of the database tables that are used in the installation of the sample application is provided. A discussion is also provided about how the Windows Installer goes about performing an installation where all the information is stored in a database. Finally there is a brief discussion of how the Windows Installer functionality can be extended along with a short description of the other types of Windows Installer packages that are used in special circumstances.

CHAPTER 4: ARCHITECTURE OF INSTALLSHIELD DEVELOPER

After the discussion in Chapter 3 about how the Windows Installer works this chapter extends the discussion to how InstallShield Developer uses this technology to create the two types of projects. The focus in this chapter is on the run-time architecture of a Standard project but the Basic MSI project is also discussed. An important section is how the InstallScript engine permits a much greater flexibility in extending the functionality of the Windows Installer than is available with the technology itself.

CHAPTER 5: CREATING PROJECTS IN THE IDE

The focus of this chapter is to create the installation packages for the sample application but this time not using the Project Wizard. This opens up many more important issues that need to be discussed relative to the creation of an installation package. Many areas of the InstallShield Developer Integrated Development Environment (IDE) are discussed in detail as the installation program for the sample application is created using a Standard project. The differences between creating a Standard project and creating a Basic MSI project in the IDE are described and then the creation of a Basic MSI project is left to the reader as an exercise. Both types of projects are included on the CD-ROM at the back of the book.

Part II: InstallScript

CHAPTER 6: VARIABLES & DATA TYPES

This chapter starts the discussion of InstallScript and covers all the data types available in the language. There are a number of examples that show how to create and use variables of these data types.

CHAPTER 7: EXPRESSIONS & STATEMENTS

The discussion of InstallScript is continued in this chapter with a description of how to create expressions and to use these expressions in statements. Chapters 6 and 7 provide all the basics that are necessary for creating installation programs.

CHAPTER 8: FUNCTIONS

This chapter covers the three types of functions in InstallScript, built-in functions, event handlers, and user-defined functions. The main focus is on the creation of user-defined functions and it is explained how to create script libraries and how to make the functions incorporated in a script library available through the Function Wizard. At the end of the chapter there is a discussion regarding the calling of functions that are in a dynamic link library.

CHAPTER 9: EXCEPTION HANDLING & COM

This chapter shows how to extend the functionality of InstallScript through the use of COM. It is possible to create COM objects in InstallScript as long as the COM server has a registered ProgID. As part of creating COM objects it is desirable to use exception handling and the capabilities for exception handling in InstallScript are also described. The capabilities of the FileSystemObject, Windows Script Host, and the Windows Installer automation interface are discussed in detail. Reference is also made to other objects that can be created but no detail is provided on these.

Part III: Getting Down to Business

CHAPTER 10: COMMON INSTALLATION TASKS

There are many common tasks that all installations need to perform. These include creating file associations, making and removing registry entries, creating environment variables, working with initialization files, and searching for installed applications. InstallShield Developer provides efficient tools for creating the functionality so that an installation can perform these operations. There are detailed examples for performing each of these tasks.

CHAPTER 11: INSTALLSCRIPT CUSTOM ACTIONS

In this chapter it is shown how to use InstallScript to extend the capability of the Windows Installer through the creation of custom actions. Custom actions can be used in both Standard projects and Basic MSI projects. The special capabilities of InstallScript for creating custom actions is explained and demonstrated. Examples are shown about how to access the database while the installation is running and how to programmatically create deferred custom actions.

CHAPTER 12: USER INTERFACE BASICS

One of the major differences between a Standard project and a Basic MSI project is how the user interface for an installation is created. In this chapter the basics of how Windows creates and handles dialog boxes is discussed. Then several dialog boxes are created in a Standard project along with the requisite dialog function written in InstallScript. The chapter then moves onto the creation of a user interface in a Basic MSI project showing the differences between it and the implementation of dialogs in a Standard project.

CHAPTER 13: INTRODUCING COMPONENTS

The proper creation of components is one of the most important and least understood of all the tasks in the creation of an installation package. This chapter discusses the rules for creating components and why Microsoft has defined the rules the way they are. Then the different tools in InstallShield Developer for aiding in the creation of components are described. Finally the chapter goes over some special issues that concern components such as System File Protection and DLL redirection.

CHAPTER 14: CREATING SPECIAL COMPONENTS

The tools discussed in Chapter 13 are put to use in the creation of various types of components such as COM, ODBC, NT services, and fonts. The subject of merge modules is introduced for creating components that can be redistributed.

Support

InstallShield Software Corporation provides the following Web site for obtaining the password for the evaluation copy of InstallShield Developer that is on the CD-ROM at the back of the book.

<http://www.installshield.com/ispres>

At this same location you can provide feedback about the book or asks questions. All input that can help make the next version of this book better are welcome.

Acknowledgements

First, I need to thank Viresh Bhatia, CEO of InstallShield Software Corporation, for taking the plunge into the InstallShield Press publishing venture. Viresh has provided me with unstinting support for this project and I am appreciative for the trust that he has placed in me.

I also need to express my thanks to David Thornley, Mingbiao Fei, and Art Middlekauff for taking the time to discuss issues related to the design of InstallShield Developer and the workings of the Windows Installer. When writing a book it is very valuable to have a sounding board off of which to bounce ideas.

I am indebted to the genius of Melvin Grefalda for the excellent graphics work that he did on the cover design. Melvin also was able to turn my crude line drawing sketches into something worthy of being displayed as figures in the book.

Finally, I need to thank the team of reviewers here at InstallShield Software Corporation for the hard work they put into going over my first draft. This work was extra on top of their normal duties. Lori Erickson performed the complete editorial review of the book. Art Middlekauff, David Thornley, Mingbiao Fei, Kent Foyer, Richard Aquino, Marwan Tabet, Michael Marino, Martin Markevics, and Gomathi Rajesh performed the technical review. These people form a major component of the development team for the InstallShield Developer product.

Part I

The Fundamentals

Chapter

1

Installation Development

The InstallShield Software Corporation has been at the forefront of the software installation business ever since the release of Microsoft Windows 3.0. InstallShield's founders recognized that creating installations for Windows applications was going to be a challenging task for developers. This vision has been borne out especially since each new release of the Microsoft Windows operating system is more complex than the last. Accordingly, for more than a decade, InstallShield has focused on providing tools for developers that facilitate the process of installation development.

This chapter introduces the subject of software installation and describes at a high level the past and present tools available to create installation programs or installation packages. The remainder of this book delves into the basics that all setup developers need to know. This book can best be described by the phrase “the basics in detail”.

Who Is Interested in Software Installation?

Three separate groups have an interest in the topic of software installation. These three groups, which are listed below, have distinctly different perspectives.

- The end user
- The LAN administrator
- The setup developer

The End User

When considering the end user's viewpoint, remember that the installation of an application is the end user's first experience with the product. A bad installation experience can create a negative impression about the product in the customer's mind before they even run the application. The end user's ideal situation is where an installation happens by itself without the need to do anything except run the application after it is installed. The reality today is much different. End users face long and complex installation processes. Many times they are asked to answer questions during the installation for which they do not know the correct answer. This can result in frustration if end users have to guess which information should be entered during the installation process. The result might be an installation that fails to do the job; the application fails to run properly or it does not run at all.

If an application fails to run after it has been installed, it is usually not the end user's fault. Most installation failures stem from a conflict between different versions of the same file that have been installed by two different applications or from an installation program that was not created correctly and cannot complete successfully. Regardless of the reason for failure, the end user can easily become an unhappy customer. Software installation technology continues to move toward making software installation more transparent to the end user. Whether installation programs will ever be so transparent that the end user is not even aware that an installation process is running is unknown at this point. A completely transparent installation can ultimately

lower technical support costs. A reliable and successful software installation for every customer removes one possible source of technical support problems and helps to keep customers happy.

The LAN Administrator

The main concern that LAN Administrators have is how to deploy software to many desktops in an organization without having to physically go to each desktop and run the installation program. The more automated this software deployment becomes, the lower the cost of maintaining numerous desktops in a complex network environment.

The challenges facing the LAN administrator are:

- Managing software installation from a central location.
- Modifying the installation program so only the necessary features are available to the end user.
- Permitting the person at the desktop to run the installation even if they do not have local system account privileges.
- Determining from a central location whether an installation has completed successfully or if there is a problem.

To provide the functionality that a LAN administrator needs, installation programs must be created to support central point deployment. Creating installation programs that facilitate easy deployment in a network environment is possible. This book provides all the basic information necessary to understand how to create these types of installations.

The Setup Developer

Many of the challenges faced by the setup developer today are the same as those faced by anyone who develops software. In many businesses, the person creating the setup has other responsibilities. Because of this, setup is often left until the very end of the development cycle, when it is time to ship the product. In some companies,

management might not recognize the complexity involved in creating a quality installation program.

Because most installation programs today are script based, the setup developer faces the challenge of maintaining scripts that might have been written by someone else. Since code documentation in the form of comment statements or specifications may be few and far between, script code becomes more difficult to maintain as time goes on.

There is a special challenge for setup developers who work in large organizations where software development takes place in different parts of the country or around the world. The challenge is to create an installation when all the parts of an application are shipped to one location where the setup developer is working. Putting the different pieces of the application together properly, so that the application works after it is installed, can be problematic.

Now that we have looked at the various viewpoints of the people who have an interest in software installation, we can discuss what is involved in the creating an installation program. The rest of this chapter provides a high-level view of installation development. Specific basic installation development topics are covered in the remainder of the book.

What Operations Constitute a Typical Installation?

In its simplest form, an installation does the following three operations in order:

1. Copies files to the target machine.
2. Registers the application with the operating system.
3. Exposes the application to the end user so that they can launch it.

It would be nice if the installation process were this simple, but it usually isn't. These three steps may have been adequate for a simple installation on Windows 3.x, but not on the 32-bit operating systems we use today.

A more complete list of operations that could compose a typical installation is:

1. Check the operating system to see if it meets the requirements of the application.
2. Check to see if other required applications are present on the target machine.
3. Verify that there is enough disk space on the target machine to accommodate the application files to be copied.
4. Present an interface that allows the end user to indicate the installation location, select which features should be installed, and provide other information (a serial number, for example). This user interface needs to solicit any information from the end user that is necessary for the installation to complete successfully. In addition, the user interface should provide feedback to the end user that indicates the installation's progress. Finally, the user interface should inform the end user about whether the installation was successful.
5. Copy files for the application features that the end user has selected for installation.
6. Make all required registry entries for the copied files to be able to function as designed.
7. Create the necessary shortcuts so the end user can run the application from the Start\Programs menu.
8. Provide a capability that allows the end user to maintain or upgrade the application, or completely remove it from the machine.

This list is significantly longer than the first and it indicates that even a typical modern-day installation has to address many different issues. The items in the second list are discussed in more detail in the following sections.

Note on terminology:

An application is composed of the application's features. A feature is a part of an application's total functionality that an end user recognizes and may decide to install independently. For example, a feature can be a spellchecker or a thesaurus. An application's hierarchy of features provides its logical definition. The building blocks of features are components, which are the atomic units that provide features with their functionality. A component is a piece of the application or product to be installed. Components can consist of single files, a group of related files, registry entries, shortcuts, installation logic, etc.

Checking the System

When creating an installation program, you may need to make sure that the operating system environment is adequate for the application to run correctly. This can be as simple as checking the major version of the operating system on which the application is being installed. Checking the operating system environment can also include checking for the service pack level on a Windows NT/2000 machine, as well as a check of both the major version number and the minor version number. It may also be necessary to check the build number of the operating system. For example, you would need to do this if you wanted to distinguish between the first (Gold) release of Windows 95 and the OSR2.5 release of Windows 95.

This type of operation is performed at the very beginning of an installation. If the check of the necessary operating system parameters succeeds, the installation moves on to the next step. However, if the check fails, the installation must be terminated. The termination process needs to provide a message to the end user detailing what went wrong and should tell the end user what needs to take place before the installation can be attempted again.

Checking for Required Applications

It is often the situation that in order to install an application, another file needs to be on the target system prior to installation. This might not be an application, but maybe a driver or a service that must be present before an application can be installed. In

other cases, you may want to create an installation for a competitive upgrade of your product and need to check that the end user has the competing product installed.

Checking for the presence of another application normally requires a search of the files on the system or a search in the registry for a particular key or value. Searching for other applications on the target system can be a fairly complex operation, particularly if you do not know where the other application may have been installed. If the installation finds that a required application is not present, the installation can initiate an installation of the required application as a child installation before proceeding with the main application installation. This is unlike the operating system check, where there is no option to programmatically update the operating system and then continue with the installation. In the operating system scenario the end user needs to update the operating system as a separate operation before the application can be installed.

In more complex situations, it might be necessary to reboot the target machine after performing a child installation. A reboot is necessary if it is the only way to start a driver or service that is required to correctly implement the installation of the main application. An example of this is where DCOM has to be installed on Windows 95 before the application installation can be successfully completed.

When checking for a competing product in the competitive upgrade scenario, the installation may find that the end user does not have the competing product installed on the target system. In this case, the installation needs to check for an identifying file on the distribution media of the competing product. It would not help customer relations if the end user had to install the competing product in order to install your application. Your installation also needs to be able to terminate itself if the end user could not produce the competing product.

Checking Available Disk Space

A typical function of an installation is to check that there is enough space on the target machine to hold all the files that will be copied. It would frustrate the end user if the installation partially completed and then the operating system displayed a message informing the end user that the disk was full. Files occupy the most space on a system, but they are not the only things that take up space when an application is installed. All but the simplest applications make many entries into the Windows registry, and these entries increase the size of the registry. An installation also creates

shortcuts that take space, along with entries that are made in initialization files that already exist, and new initialization files that are created during the installation.

Calculating the space required to install an application is a complex operation. To do this calculation properly, the check for available disk space must take into account the difference in size between the files that are being copied, and files with the same name and location that are being overwritten. When a situation arises during an installation where the installation program finds that there is not enough space to install the application, the installation program needs to offer the end user the opportunity to choose a different installation location. In fact, wherever there are roadblocks to completing an installation successfully, a good installation program should try to find alternatives or let the end user make different decisions.

The User Interface

The user interface leads the end user through the installation process and provides feedback to the end user regarding the installation's progress. These two functions are made up of the following:

- The wizard that asks the end user the questions that need answering before the installation can be completed.
- The progress dialog that shows the end user how the installation is progressing, along with the dialogs that show any errors that occur during the installation.

A good installation program provides default values for all questions. If the end user does not know how to correctly answer a question, they can accept the default value and complete the installation successfully.

The Wizard

A typical user interface wizard for an installation provides the end user with a license agreement dialog, a dialog that collects the user name and company name, and a dialog that allows the end user to select the application features to be installed. Many applications, however, require a much more complicated user interface.

In the dialog that displays the license agreement, the user has to indicate agreement with the license terms or the installation is terminated. The dialog that asks for the user name and company name can also require the end user to enter a serial number or other type of security-related entry. If the end user does not make the correct entry, the installation is not allowed to continue.

The dialog that presents the available setup types to the end user usually allows the end user to select from predefined setup types or permits them to customize the installation by selecting the features that they want installed. One of the predefined setup type options is the default for the installation.

The Progress Dialog

Feedback to the end user during the installation is generally provided via a progress bar that is annotated with a description of the current actions. Some installations display a progress bar that indicates the progress of each individual action and one that shows the progress for the entire installation process. Part of the feedback mechanism is to notify the end user of any errors that occur during the installation that make the installation incomplete. Another part of this feedback mechanism is a final dialog that informs the end user that the installation process is finished and whether it was successful.

There are cases where you might want to install an application without presenting any visible sign to the end user that the installation process is running. This is a silent installation and it is used primarily in a networked environment where software is placed on the desktop from a central location. The ability to run an installation silently is an important consideration in most installation programs.

There are now options between a completely silent installation and an installation that provides a full user interface. This in-between type of user interface usually involves only the display of installation progress or notification that the installation is complete.

Copying Files

The topic of copying files to the target system may seem relatively straightforward. However, there are several issues that come into play when files are being copied. One of these is: What rules are used when a file of the same name already exists on

the target system? When the installation program is created, the setup developer has to decide when the installation should overwrite a file on the target system and when it should not. Another issue arises when a file that the installation is copying to the target system already exists on the system and is being used by another application while the installation is in progress.

There are other lesser-known issues surrounding the operation of copying files during an installation. These issues will be discussed throughout the rest of this book.

Registering the Application

The true complexity of creating an installation program comes from being able to make the required registry entries so that the application works correctly after installation. Most modern applications use the Component Object Model (COM) to create the functionality that the end user sees. COM requires heavy use of the registry, and all the required registry entries are made during the installation, after the files have been copied. The registry is also used to enable database connectivity, to provide information about where certain files are located on the system, to associate file extensions with an application, and many more things that are required for an application to be able to work correctly.

The registry is also used to enable the removal of an application, as well as to allow the application to be upgraded. Much of the discussion in this book revolves around how the registry is used. Many of these registry related topics will be discussed in depth in later chapters.

Making the Application Available

This operation allows the end user to easily access and launch the installed application. It consists primarily of creating shortcuts on the Programs menu or on the Desktop. In addition, it involves allowing the end user to launch an application, while in Windows Explorer, by double-clicking on a file with an extension that has been registered to the application.

With the advent of Windows 2000, a new method for making an application available was introduced. This new method is called Advertisement. Advertisement is a mechanism that makes the application appear to be installed when only registry

entries have been made, but no files have been copied. The application appears to be installed because there is a shortcut to the application on the Programs menu. When the end user tries to run the application for the first time from the shortcut, the application is installed and run.

Maintaining The Application

After an application has been installed and is working correctly, the end user may want to modify the installed application. There are three types of maintenance operations that can be performed on an installed application:

- **Modify:** Allows the end user to add new features not originally installed or remove features that are no longer wanted.
- **Repair:** Allows files for installed features to be reinstalled. This is done if it is suspected that files have been deleted by mistake or have been corrupted.
- **Remove:** Allows the application to be uninstalled.

All good installation programs provide the capability to perform maintenance operations for an installed application.

The preceding sections discussed the typical operations that are performed during a normal application installation. The next section examines the various installation types. It is important to understand that removing an application is considered just the reverse of an installation. When we talk about installation, we are also talking about uninstallation.

What Are the Various Types of Installations?

This section introduces the various types of installation programs that can be developed, and describes their characteristics. In the following subsections, eight different types of installation programs are defined. We will only look at a few of

these eight types of installation programs throughout the book. The next section provides definitions of these different installation programs.

The Fresh Install

This type of installation package is what was discussed in the previous section. This is where an application is installed for the first time and has not already been installed on the target system. If the application has been installed previously, it is uninstalled at the time the fresh install is performed and is used to perform a major upgrade of an application. This type of fresh install is discussed in The Upgrade Install section.

The Maintenance Install

This type of installation occurs only after a fresh install has been performed. It is performed using the same installation program that was used to perform the fresh install. As previously mentioned, a maintenance install allows the end user to change the feature set of an installed application by adding new features or removing installed features. The end user also has the option to repair what was previously installed or to completely remove the application from the system. The important thing to remember about the maintenance install is that you cannot use this type of install program to add new features to the installed image that were not defined in the original installation program.

The Advertised Install

In general, an advertised install is where an application is made available to the desktop computer from a central location, but is not actually installed until the end user takes specific action to install the application. It is a pull mechanism where it is the end user that initiates the installation process. This type of install is part of the Windows 2000 deployment functionality and requires the use of the Windows Installer technology to create the installation program. The Windows Installer is a new service provided by Microsoft that enables the creation of more robust installation programs. The basics of this new technology are introduced in Chapter 3.

An advertised install is implemented as either a published application or an assigned application. A published application is available to the end user from the

Add/Remove Programs applet: Running the installation program from this applet installs the application. A published application can also be installed when the end user tries to open a file that is served by the published application.

An application that is assigned displays a shortcut icon on the Programs menu. The end user forces the installation when they attempt to launch the application for the first time using this shortcut. There are two types of assignment; assignment to a specific user and assignment to all users of the machine. When assignment is made to a particular user, the application is launched when that user tries to run the application. When assignment is made to all users of the machine, the application is installed the next time the networked desktop computer is booted.

The important thing to remember about an advertised install is that it is part of the Windows 2000 deployment mechanism. This mechanism can be implemented only on a pure Windows 2000 network. Note that this is a pull technology. The central LAN manager makes an advertised application available to the desktop, but the actual installation process is initiated from the desktop itself.

The Administrative Install

An administrative install is a means to implement a pull mechanism without requiring a pure Windows 2000 network. The administrative install is simple in concept. Its purpose is to define the image of an installation program on a network drive. End users need to navigate to the network drive location and run the installation from that location. The only actions that occur when the administrative install is run are uncompressing any files that were compressed as part of the installation program and copying these files to the network drive location. No registry entries are made and no shortcuts are created. Compressed files are uncompressed for the purpose of allowing a patch operation to upgrade the application.

The Upgrade Install

An upgrade install is used in cases where an application has already been installed on the target system and a new install is run in order to upgrade the application to a newer version. This type of install is implemented using two different approaches depending on the degree of change between the earlier version of the application and the newer version.

If the difference between the earlier application and the newer application is small enough that there is only a minor version change, simply installing the newer version of the application over the older version performs the upgrade. The files that have changed are replaced with their corresponding newer versions. This is the reinstallation approach to performing an upgrade install.

When the difference between the earlier application and the newer application is significant enough to require a major version change between the applications, the approach to upgrading the application changes. In this situation, a two-step process is wrapped into one installation program. These two steps are:

1. Silently uninstall the older version of the application.
2. Install the newer version of the application.

This is the fresh install approach to performing a major application upgrade. The key is to find where the earlier version of the application has been installed so it can be uninstalled before installing the newer version. It is possible to perform the above steps in the reverse order, but this could lead to problems if your installation has been created incorrectly.

The Patch Install

The patch install is a special type of install that performs an upgrade of an installed application. The patch install is useful because it has a smaller upgrade program size than that afforded by the upgrade install discussed in the previous section. Patching is a mechanism where only the bits that are different between two application files are shipped as part of the upgrade, instead of the complete new file.

When the patch is applied to the earlier application file, it is turned into the newer application file, changing out those bits that are different between the two files. A patch install can be used regardless of whether there is only a minor version change between the two applications or there is a major version change.

The Transformed Install

This type of install is a special operation that can be executed with an install that uses the Microsoft Windows Installer technology to create the installation program. This type of install begins with a fresh install program and then, when the installation program is launched, a transform is applied to the installation program. The effect of this transform is to modify the program so it is different than what is contained in original fresh install package. The use of a transform allows the LAN administrator to take a large installation program and modify it temporarily at install time. For example, the number of features that are available to the end user can be modified by the transform. With the transform mechanism, the end user can install only the features that are necessary for their specific job function.

Note that a transform is simply a representation of the difference between two different installation programs. When this transform is applied to one installation program during the install, it changes it into the other installation program. This is done only in memory so no permanent changes are made to the installation program that is being launched. Transforms can also be applied to an advertised install so that only the modified package gets installed when the end user tries to launch the application.

The Nested Install

To launch one installation program from another, you can use a nested install. A nested install runs the installation of a third party application as part of an installation program. This need arises when an installation program checks for other required applications on the system and does not find them. In this case, the installation program would run the installation programs for these missing applications as part of the main installation program. This way, the end user is not requested to install the required applications separately before being allowed to run the installation for the main application.

Installation Development Technologies and Tools

As discussed in the previous section, creating an installation program is no simple task. To create an installation program from start to finish using your favorite programming language would require significant effort. InstallShield Software Corporation produces installation development tools that help reduce the considerable time investment that was required to create an installation program. InstallShield's installation development tools make creating installation programs much easier and handle some installation tasks without any involvement from the setup developer.

Until Windows 2000 was shipped, installation tools had to support only one technology. This technology was a script-based approach to performing installations. With Windows 2000, Microsoft introduced Windows Installer, which is a database-oriented technology. The development tools that support this new technology are dramatically different than those used to support the development of script-based installation programs.

The next two sections take a high-level look at the two approaches to creating installation programs. At the same time, this section briefly discusses the InstallShield development tools that have been created to support these installation technologies. The final section of this chapter describes the InstallShield Developer product, which combines the flexibility of the script-based approach to creating installation programs with the capabilities of Microsoft's Windows Installer installation service.

Script-Based Installation Programs

The creation of the original InstallShield installation development tool was based on a scripting language that is called InstallScript. Over the years this scripting language has been enhanced continually until it now supports advanced operations such as creating COM objects and exception handling. The benefit of InstallScript is that it provides many built-in functions that perform installation-related operations that would require considerable effort if the same functionality were created using a standard

programming language. InstallScript also supports the creation of user-defined functions when the setup developer wants to add custom functionality.

Versions up through InstallShield3

Through the release of the InstallShield3 product, all of InstallShield's installation development tools were command line-based. This means that the script had to be compiled at the command line, and the files that made up the application were compressed into a library using a separate utility. At this time, most software was distributed on floppy disks. The compression utility had to split the application files into separate libraries that would fit on a floppy disk. Because of the limited space on a floppy disk there was a continuing search for a better compression algorithm, one that would minimize the number of floppy disks required for distribution.

InstallShield3 was the first InstallShield product that supported the 32-bit environment introduced with the release of Windows 95. The transition from 16-bit systems to 32-bit systems was a complicated time for installation development tools because there were many different environments that could be targets of an installation. Windows 3.x still had to be supported, as well as Win32s. (Win32s was a short-lived attempt by Microsoft to allow programmers to start programming using the 32-bit Windows APIs but still distribute these applications to Windows 3.x.)

During this same timeframe, it was also necessary to distribute applications to three other platforms that used different processors. These were the DEC Alpha, MIPS, and the Power PC. All of these platforms ran a version of Windows NT. Now, however, you only have to be concerned about the Intel platform running some 32-bit version of Microsoft Windows. Of course you still have to take into account the differences between the Windows 9.x and Windows NT operating systems.

When an installation program created with InstallShield3 is run, it needs to place a number of temporary files on the target system. These temporary files are used to run the installation, with the most critical file being the scripting engine. The scripting engine parses the compiled installation script and performs the installation functions as defined therein. When the installation is complete, these temporary files—including the scripting engine—are removed from the target system.

During the installation, a log file is created that allows for the proper removal of the installed application. A special executable is left on the target system and this

executable is used to perform the uninstallation. This executable reads the log file to determine what files need to be deleted and what registry entries need to be removed. Any operation that is not logged during the installation will not be removed during the uninstallation. The setup developer can control what is logged and what is not by making the proper entries in the installation script.

The replacement for the InstallShield3 product was the InstallShield 5.x series of releases. The release of InstallShield 5.0 did away with the command line-only mode of operation and provided the setup developer with a user interface.

InstallShield Professional 5.x

The development of the InstallShield3 development tool focused on providing an installation experience that would be better for the end user. This included the Windows 95 style of dialogs and defaults that met the Windows 95 Logo requirements. The design focus for InstallShield 5.x was to provide the setup developer a friendlier development environment. This included a built-in script editor, a visual resource editor, a visual registry editor, and a better way to control media images that were created.

For the first time, an InstallShield installation development tool was given a project basis similar to what exists in Visual C++ or Visual Basic. This meant that everything the setup developer entered into the Integrated Development Environment (IDE) of InstallShield 5.x was stored in an installation project. The build process launched from the InstallShield 5.x IDE turned the entries in the installation project into a media image that was then copied to the distribution media.

The main change on the media image from that created by InstallShield3 was that a new compressed library format was used. This compressed format had a .cab extension that caused some confusion, since this was not the same file format as a Microsoft cabinet file even though the same file extension was used.

The scripting engine was slightly modified from what was used with InstallShield3 because of some new functions added to InstallScript. However, the implementation was essentially the same. Also, a log file was created for performing the uninstallation, and the executable that read this log file was left on the target system in the same manner as InstallShield3.

After the InstallShield 5.x series of releases came InstallShield 6.0. This product provided a completely new compiler, scripting engine, and objects that became a means to encapsulate the installation logic for specific technologies.

InstallShield Professional 6.x

The InstallShield 6.0 release introduced a much more robust scripting language that included new data types, the capability to create COM objects, exception handling, and additional smaller advances for making script-based programs more robust. A new scripting model was also introduced. This new scripting model was called the event-driven model. It was still possible, however, to create scripts that used the older procedural model. The two types of scripting models are discussed in more detail in the next section.

The release of InstallShield Professional 6.0 presented a new means for maintaining an application once it was installed. Prior to this release, the only maintenance that could be performed on an installed application using a script-based install was to totally uninstall it. Now it was possible to provide the complete range of maintenance operations that consist of modifying the original installation of the application, repairing the installation, and removing the installation. To enable this capability, it is necessary to leave on the target machine all the files that make up the original installation. This includes the installation bootstrap executable SETUP.EXE, the log file that records what was initially installed, the compiled installation script, and the files that provide the maintenance operation user interface.

InstallScript Execution Models

Until the release of InstallShield Professional 6.0, there was only one script execution model called the procedural model. This model consists of three distinct blocks of code that appear in any installation script that is created. This model is still supported by InstallShield Professional 6.x. The three function blocks are defined as follows:

```
// The DECLARE BLOCK is where constants are defined,  
// global variables declared, functions are prototyped,  
// and header files are included as shown in the  
// following example.
```

```
#define MAX_LENGTH 260 // declare constant
```


PART I THE FUNDAMENTALS

```
STRING    szTitle, szMsg; // declare global string variables

prototype DisplayMsg(STRING, STRING); // function prototype

#include   "isrt.h" // include a header file

// The PROGRAM BLOCK is where the control of the
// installation is handled. This is like the main()
// function that is the entry point for console
// applications.

program

    // Assign value to global variable
    szTitle = "InstallScript Test";

    // Assign value to global variable
    szMsg = "This is a test message";

    // Call function to display message
    DisplayMsg(szTitle, szMsg);

endprogram

// The FUNCTION BLOCK contains the definition of
// all user-defined functions that are called as
// part of the installation program.

function DisplayMsg(Title, Msg)
begin
    // Call a built-in function to display the message.
    sprintfBox(INFORMATION, Title, Msg);
end;
```

As you can see by looking at the above code snippet, this looks like a standard structured programming approach to creating an installation program. And that is exactly what it is, a standard linear approach.

With the release of InstallShield Professional 6.0, a new event-driven scripting model was provided, in addition to the procedural model described above. The difference between the two models is that the new event-driven model has an implicit program

block in which it calls a predefined set of functions. Some of the functions that are called are the built-in functions provided by the InstallScript language. Other types of functions are called event handlers and it is inside these event handlers that you add InstallScript code in order to create an installation program.

InstallScript defines the available event handlers and all of these event handlers have function names that begin with the string "On". Event handler names include `OnBegin`, `OnFirstUIBefore`, and `OnFirstUIAfter`. To program successfully with the event-driven model, you need to know when each of these event handlers is called. Only then do you know into which event handler you need to insert your InstallScript code to perform the actions that are required for the installation.

When you use the event-driven model, your code might look like a bunch of individual functions without any entry point. The entry is provided to your script by the compilation and linking process. In fact it is the design of the linking process that allows both the procedural and the event-driven model to be used. The event-driven model is the default when creating a new project in InstallShield Professional 6.x, but if the script already contains an explicit program block then the linker replaces the implicit program block with the explicit program block. It is this functionality of the linker that allows scripts created in InstallShield 5.x, which have an explicit program block, to also be compiled.

After this short introduction to script-based installations, we can now move on to a discussion of the new technology for creating installation programs. Microsoft developed this new technology and shipped it with Windows 2000. By the time you finish this book, you will have a very clear idea of both ways to create installation programs.

Windows Installer-Based Installation Programs

With the Windows Installer, Microsoft has given setup developers a new approach to generating installation programs. They provide the installation engine and all setup developers have to do is author the installation package according to the guidelines specified by Microsoft. The installation engine is part of the operating system on Windows 2000 and Windows Me, and can be installed on Windows 95, Windows 98,

and Windows NT 4.0. The term "installation package" can be used to refer to a Windows Installer-based installation instead of "installation program". This is because the Windows Installer technology is a data-driven approach to performing installations. In this approach, the setup developer has to populate a database that provides all the information that the Windows Installer engine needs in order to perform the desired installation.

The development tool that is required to support this new technology has to be quite different from the type of tool that was developed to create script-based installation programs. The development tools that InstallShield has created to work with the Windows Installer technology are authoring tools. They are called authoring tools because they abstract most of the information that needs to be placed into the database by prompting for values in an understandable fashion. The responses to these questions are saved in a project file. During the build process, the information in the project file is used to make the proper entries into the Windows Installer database.

This database, along with the files that make up the application, forms the installation package. When this installation package is passed to the Windows Installer engine, the engine reads the database and performs the installation. Because of the newness of this technology, there have been only two major releases of an authoring tool that supports the Windows Installer concept. After discussing the benefits offered by the Windows Installer technology these two major releases are briefly described.

Why Use the Windows Installer Technology?

The Windows Installer technology had its roots in the Microsoft Office team where there was an initiative to solve the problems that users were having with failed installations and uninstalls. Problems revolving around these two operations were generating a significant load on the technical support organization. In order to reduce this expense a new approach was created for performing the total management of an application's lifetime on a machine. The Windows Installer was felt to be of such a significant value that it was made available to all developers of software by making it part of the Windows 2000 operating system.

An installation created using the Windows Installer technology is able to work with the new software deployment capabilities that are included with Windows 2000 and later. These capabilities permit authorized users to be able to install software even if

they do not have administrative privileges. The Windows Installer also provides a capability where applications can be created so that they have the ability to repair themselves.

The Windows Installer supports all the new concepts that are produced by Microsoft on a continual basis. For instance there is now a 64-bit version of Windows XP Professional and the unique requirements for installing software to this system are fully supported by the Windows Installer. This is also true for the new capabilities for restoring the system to previous states that are available with Windows ME and Windows XP.

There are many compelling reasons that installations should be created using the Windows Installer engine for making the changes to the target operating system. Not the least of these reasons is that in order to obtain the “Certified for Windows” logo the application has to use the Windows Installer for its installation. Chapter 3 provides more detail into the design of the Windows Installer technology.

InstallShield for Windows Installer

This is the name of the version 1.x of the Windows Installer authoring tool that was released. This product required a fairly good knowledge of the Windows Installer technology in order to create installation packages based on this technology. This product was limited in some areas and was replaced with the version 2.x releases as described in the next section.

One major feature that was implemented in this product was a built-in visual resource editor where the setup developer could construct dialog boxes in a similar fashion to the construction of forms in Visual Basic. Due to the unique approach that the Windows Installer takes to defining a dialog in the database this feature abstracted the interface to at least a half dozen tables.

InstallShield Professional – Windows Installer Edition

This is the name of the version 2.x releases of the Windows Installer authoring tool from InstallShield Software Corporation. In this product, the IDE was changed significantly from that in the version 1.x releases in order to hide more of the complexities of the Windows Installer technology. Setup developers who used this tool could still work in the same mode that they worked with in the earlier releases.

Additional enhancements were made to the visual resource editor used to create dialogs.

This release abstracted more of the data that is added to the Windows Installer database. It also provided an extensive automation interface so that the project file can be manipulated programmatically. Other added features included an interface to source code control systems, MSI validation, and an MSI debugger. However, there were still some things that could not be accomplished with this tool. Many of these deficiencies are corrected in the InstallShield Developer version 7.0. This new product, discussed in the next section, combines the best aspects from script-based installs with the best parts of Windows Installer-based installs.

InstallShield Developer

InstallShield's newest installation development tool is InstallShield Developer and it provides a development environment that allows for both the creation of script-based installation programs and the creation of pure Windows Installer packages. This latest product is the upgrade path from both the InstallShield Professional – Standard Edition version 6.3 and the InstallShield Professional - Windows Installer Edition version 2.03 products. Even though this is considered an upgrade for InstallShield Professional – Standard Edition version 6.3, the InstallShield Professional – Standard Edition product will still be developed and setup developers who now use this product can continue to count on the product being enhanced and supported. However, over the long run, the InstallShield Professional – Standard Edition product will not have the capability that InstallShield Developer has.

The one primary difference between InstallShield Developer and InstallShield Professional – Standard Edition is that InstallShield Developer uses the Windows Installer installation engine. The scripting engine that runs the compiled InstallScript still exists in InstallShield Developer, but it runs on top of the Windows Installer engine. In this way, you get the flexibility of a scripting environment and the power of the Windows Installer technology. The reasons for using the Windows Installer as the underlying install engine were touched on in the section entitled “Why Use the Windows Installer Technology?”. A detailed discussion of the capabilities of the Windows Installer is provided in Chapter 3.

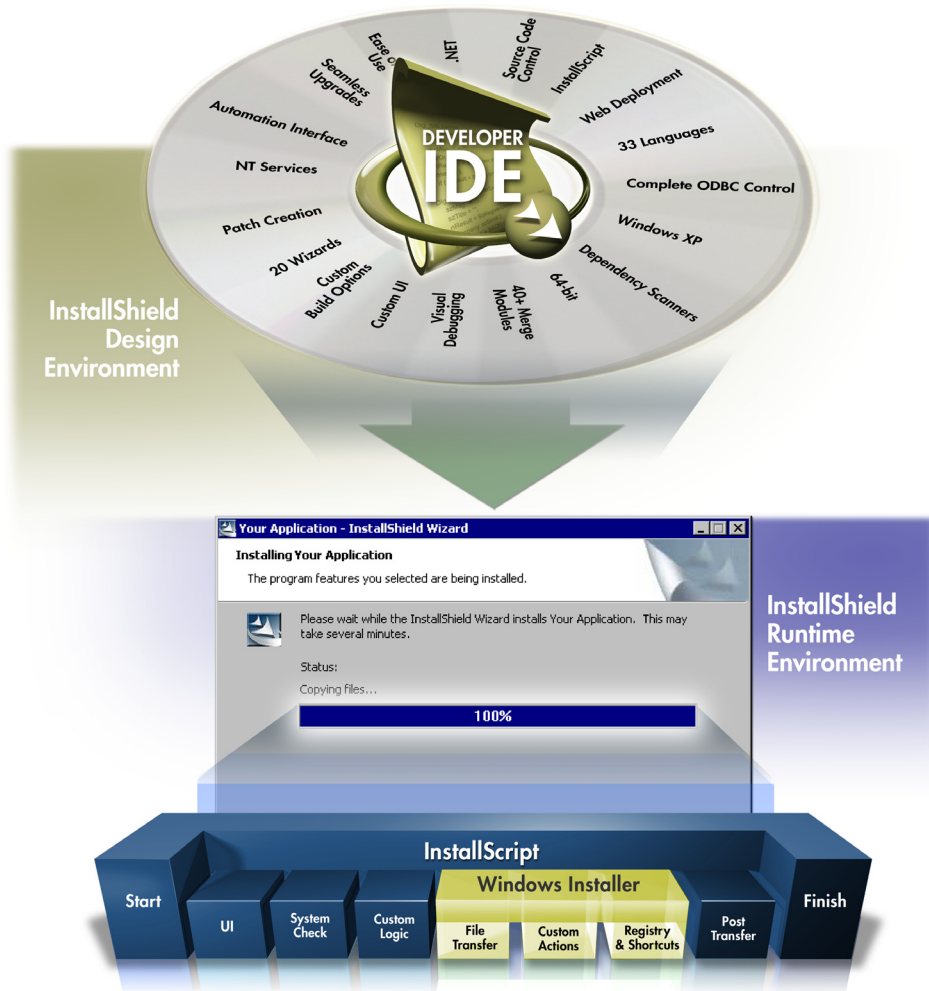


Figure 1-1: *Conceptual overview of the architecture of InstallShield Developer.*

Figure 1-1 provides a nice conceptual overview of InstallShield Developer. InstallShield Developer contains many features that have been enhanced over what they were in previous InstallShield products. There are also a number of new features.

Some of the significant new features are support for Window XP, support for .NET, and support for new 64-bit applications. The remainder of the book will discuss working with InstallShield Developer.

Conclusion

In this chapter, we have seen that creating effective software installation programs is not a simple task but it is made easier by using the correct installation development tool. To do the job correctly, setup developers need to have a solid understanding of the Windows operating system. For a long time there was only one way to create an installation program and that was through the use of scripting. Even if a development tool appeared to be only point and click, there was still a script running in the background when the installation was launched. Script-based installation programs need to include the engine that executes the script. After the installation is complete, the engine is normally removed from the target system.

When Windows 2000 shipped, it included a new technology devoted to software installation. This new technology is called Windows Installer and it is included with Windows 2000, Windows Me, and Windows XP. The Windows Installer can also be installed on Windows 95, Windows 98 and Windows NT 4.0. Setup developers working with this new technology need to author a special database that the Windows Installer reads. This database contains all the instructions that the Windows Installer needs to perform the installation. Unlike with script-based installations, the setup developer does not need to provide an installation engine since this is already part of the operating system.

Introducing InstallShield Developer

This chapter introduces the InstallShield Developer product and its capabilities. InstallShield Developer is the most powerful installation development tool that has been released. There are 20 task-based wizards that make difficult installation development tasks much easier. InstallShield Developer also has many features that enable Web distribution. You can now use InstallShield Developer to create installation programs for applications created using the Microsoft's new 64-bit API. In addition, InstallShield Developer supports the creation of setup programs that target Windows XP, as well as setup programs for applications that use the .NET technology. As mentioned in Chapter 1, InstallShield Developer combines the full power of InstallScript with the functionality provided by Microsoft's Windows Installer functionality.

Projects created with earlier versions of InstallShield installation development tools can be upgraded to InstallShield Developer projects. It is also possible to convert a Windows Installer -based project to a script-driven project.

A Quick Tour

This quick tour provides an overview of InstallShield Developer's general layout. In this section only the basic features of InstallShield Developer's user interface are covered. Detailed coverage of the specific product capabilities is provided when we discuss the various issues related to creating installation programs.

The Opening Screen

When you launch InstallShield Developer from the Programs menu, you see a Welcome dialog that offers three options (Figure 2-1).

- **Open my InstallShield Today page:** The default selection displays the Welcome View of the InstallShield Today page (Figure 2-2).
- **Launch the Project Wizard:** This option launches the Project Wizard immediately when you click OK. The use of the Project wizard is covered later in this chapter.
- **View the Product Tutorial:** The third option directs you to a Tutorial that introduces the product. The InstallShield Developer Tutorial shows how to use the Project Wizard to create a script-based project. It also introduces the Release Wizard and Component Wizard, and provides an overview of the Integrated Development Environment (IDE).

If you click the Cancel button, the Welcome View of the InstallShield Today page is displayed in the IDE. If you do not want this Welcome dialog to be displayed again when you launch InstallShield Developer, deselect the check box at the bottom of the Welcome dialog. If you later decide that you want the Welcome dialog displayed, you need to go to the registry to re-enable it.



Figure 2-1: *The InstallShield Developer Welcome dialog.*

The Welcome dialog setting in the registry is a per-user setting. If you do not want the dialog displayed, another user can still have it displayed. To re-enable the dialog:

- Open the registry editor and go to the following key:
HKCU\Software\InstallShield\Developer\7.0\Project Settings
- Under this key, find the 'Project Intro Dialog' value.
- Set the value data for this value name to 1.

After you make this change, the Welcome dialog will be displayed the next time you launch InstallShield Developer.

The InstallShield Today Page

The default selection in the Welcome dialog, the InstallShield Today page is displayed in the IDE. Figure 2-2 shows the InstallShield Developer IDE without an active project loaded.

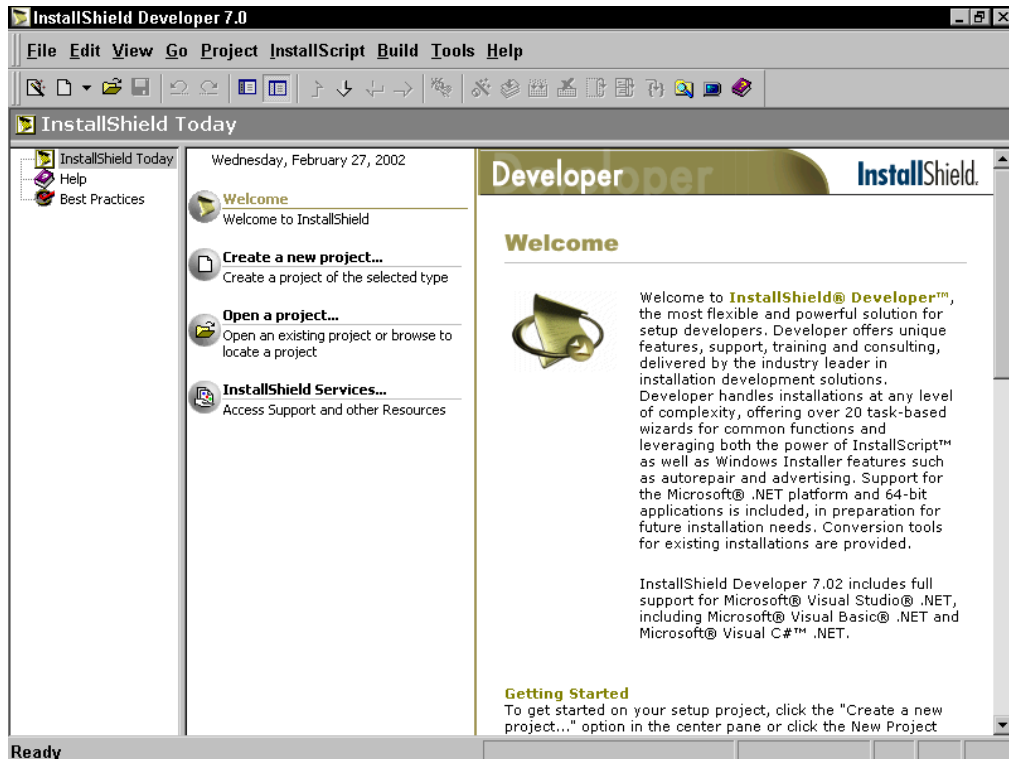


Figure 2-2: *The InstallShield Today page.*

On the left side of the screen is a vertical panel called the View List. The center of the screen contains another panel that is a sub-view list and the right hand side of the screen contains the selected view. This layout of the IDE in Figure 2-2 is similar to what you see when you have a project loaded. There is an additional vertical panel called the Viewbar that can be displayed. The Viewbar is an alternative to the View List. Normally either the View List or the Viewbar is used for navigation purposes but not both at the same time.

Within the InstallShield Today view, there are four sub-views:

- **Welcome:** This view provides basic product information, InstallShield news items, and update information. This view is updated dynamically from time to time with news about maintenance packs, top KB articles, etc.
- **Create a new project:** From this view, you can create a Standard (script-driven) installation project, a Basic MSI (Windows Installer-based) installation project, or a merge module project. You can also launch one of several wizards. Working with this view is discussed later in this section.
- **Open a project:** This view displays all of the projects that you have created.
- **InstallShield Services:** This view provides a central location from which you can easily access InstallShield support, training, and product information.

The Create a New Project View

When you want to create a new project you will most likely go to the “Create a new project” view that is shown in Figure 2-3. The “Create a new project” view provides the facilities to create a Standard (script-driven) installation project, a Basic MSI (Windows Installer-based) installation project, or a merge module project. These project types can also be created from the Files pull down menu or from New button on the toolbar. You can also take a Visual Basic project and, from it, create an installation project or you can add the contents of a Visual Basic project to an already existing installation project. For this to work, you need to have Visual Basic 6.0 installed on your build system.

Finally, from this view you can launch one of several project wizards. The C# .NET Project Wizard, the Visual Basic 6.0 Wizard, and the Visual Basic .NET Project Wizard allow you to create installation projects for your applications created in C# .NET, Visual Basic 6.0, and Visual Basic .NET respectively. These three types of projects can also be created from the Project pull down menu. For other applications, the Project Wizard directs you through the process of creating an installation project. The Project Wizard can also be launched from the Files pull down menu or from the left most button on the toolbar. The last half of this chapter explains how to use the Project Wizard to create a project.

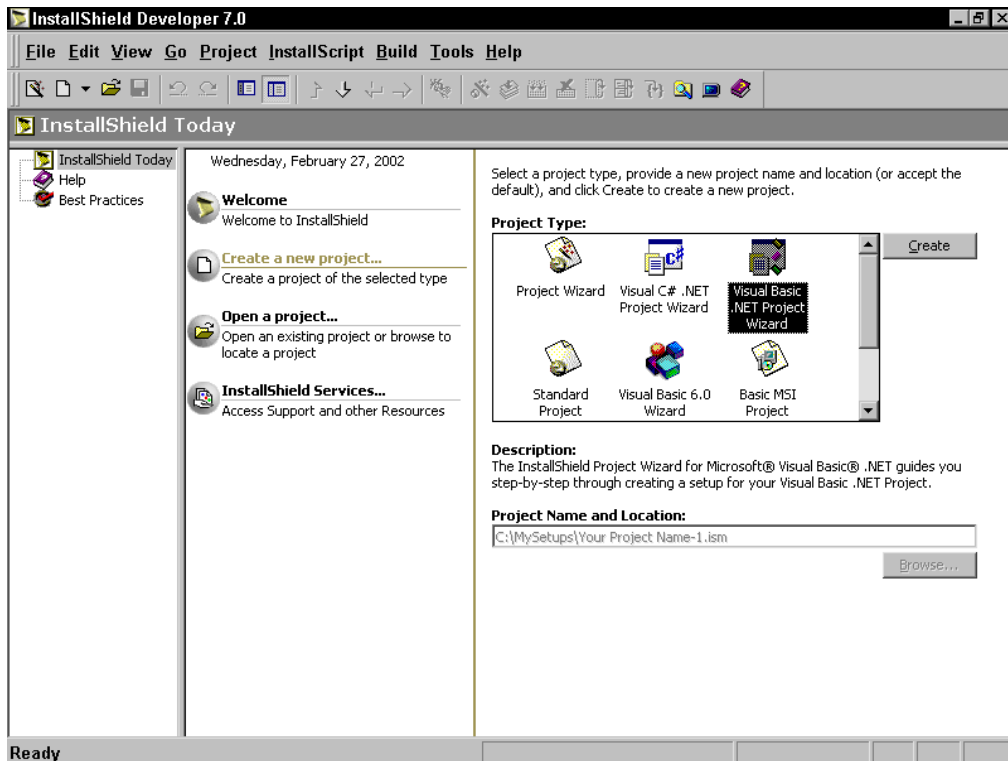
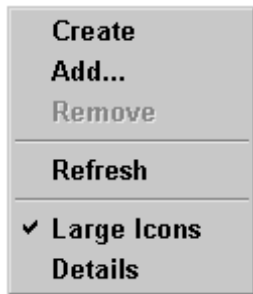


Figure 2-3: *The "Create a new project..." View*

When you highlight one of the icons in the Project Type pane, a description of what double-clicking on this icon will do is displayed in the Description section below the Project Type pane. To create a blank Standard, Basic MSI, or merge module project in this view, do the following:

1. Click the icon that corresponds to the project that you want to create:
2. Type the project name and the path you want to use in the Project Name and Location field, or click Browse to navigate to an existing location.
3. Click Create.

When you right-click anywhere in the Project Type pane, the context menu shown here will be displayed. The context menu options are described below:



Create: Launches the type of project or wizard that is highlighted in the Project Type pane.

Add...: Allows you to add a template to the icons that are part of the Project Type pane by default. Templates allow you to customize the starting point for projects that you want to create.

Remove: Removes any templates you have added to the Project Type pane. You cannot remove the default icons.

Refresh: Makes visible all templates that have been created in the Templates folder.

Large Icons: Affects how the various types of projects are displayed in the Project Type pane. This is the default display type.

Details: Changes how the various types of projects are displayed in the Project Type pane. This detailed display provides the description beside each of the project types in the “Open a project” view.

The Open a Project View

The third sub-view is the “Open a project” view (Figure 2-4). . Displayed in this view are all the projects that you have created. You will navigate to this view when you want to open an existing project.

When you highlight a project, details about the project are provided in the space below the Project List pane, including the file name, project type, the date and time of last modification, and the location. Below the project information is the “Reload the last project at startup” check box. You select this option to have the last project you opened loaded when you launch InstallShield Developer.

If there are projects that do not appear in the Project List pane, you can browse for them using the Browse button. Once you browse to a project, an icon for that project appears in this pane.

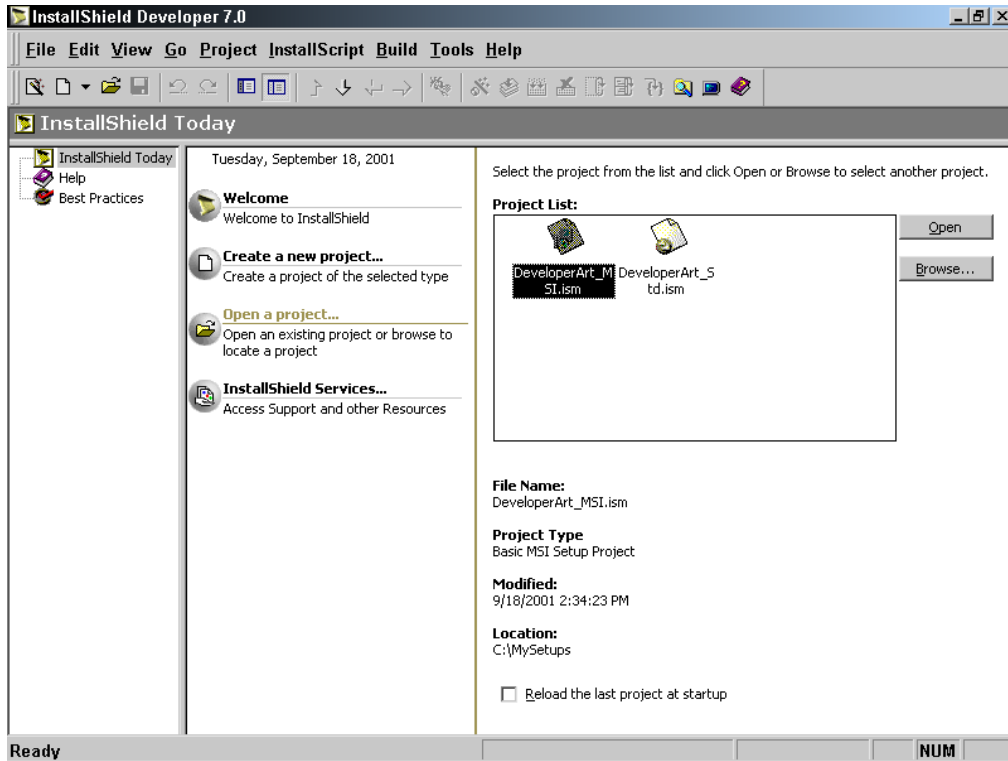


Figure 2-4: *The Open a project... view.*

When you right-click when the mouse pointer is in the Project List pane, the context menu shown here will be displayed. The context options are described below:



Open: Opens the project that is highlighted in the Project List pane.

Browse: Allows you to browse to a project file that is not shown in the Project List pane. After you have identified a project file from a new location, it appears in the Project List pane whenever you launch InstallShield Developer.

Remove: Removes any projects in the Project List pane that you no longer need.

Rename: Renames the project file.

Large Icons: Affects how the various types of projects are displayed in the Project List pane. This is the default display type.

Details: Changes how the various types of projects are displayed in the Project List pane. This detailed display provides the location of each of the projects.

Help View and Best Practices View

The Help and Best Practices views can be displayed by clicking in the View List on the appropriate icon. The Help view provides a number of links to various resources, including tutorials, the Getting Started Guide, and the online Help. The Best Practices view gives a description of some of the rules that need to be used to create components. We will discuss these rules in Chapter 3 and in more detail in Chapter 13.

The Menus

As with most Windows programs, the top of the screen contains the title bar, the menu bar, and the toolbar. The title bar provides the name of the project that is open and the project type.

This section briefly examines some of the menu items (Figure 2-5). Most of the menu items will be discussed in detail as we get to those parts of the book that use them. However, there are a few menu items we want to look at right now because they impact how you create projects.



Figure 2-5: *The InstallShield Developer default menu bar.*

This menu is a dockable menu. Using the drag point for this menu, you can drag the menu bar around and place it anywhere you want in the IDE. The drag point is the double bars at the left side of the toolbar. If you double-click on this toolbar, it becomes a floating toolbar. Double-clicking on it again docks it back in the original position. In the next few pages, the File, View, Tools, and Help menus are discussed.

The File Menu

From the File pull-down menu, you can perform the same types of operations that are available from the “Create a new project” view and the “Open a project” view.



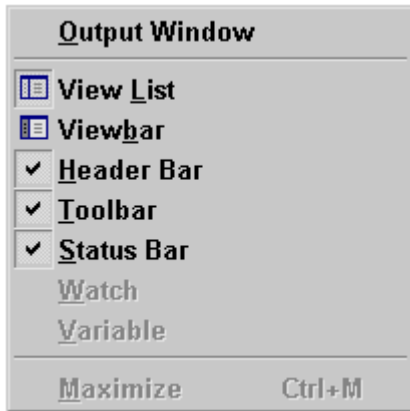
All of the options on the File pull-down menu relate to the handling of project files, except for the print commands. The print options relate only to the printing of script files. The first option on the File pull-down menu allows you to launch the Project Wizard and the second option allows you to create a new project, which can be a Standard project, a Basic MSI project, or a Merge Module project.

Using the Open command, you can open an existing project. Using this option also allows you to convert many other types of files to a project file. The Close option closes the current project and displays the InstallShield Today page.

The View Menu

The options on the View pull-down menu allow you to modify what is shown in the IDE. When you want to gain more screen area, you can turn off some of the views that typically comprise the IDE. By default, the View List, Header Bar, Toolbar, and Status Bar are displayed. Select and deselect various options on this menu to see each of these items in the IDE.

At the top of the View menu is the Output Window option. The output window is where feedback is displayed when you perform a build operation. The Output



Window option allows you to return to the last output after you have closed the window.

The Watch and Variable options relate to the MSI Debugger, and are enabled only for Basic MSI projects. The final option on the View menu allows you to maximize the Script Editor or the Dialog Editor to cover the total width of the IDE. This option is enabled only when the focus is in the Script Editor or the Dialog Editor. You will have the opportunity to use this option when we cover InstallScript and

when we cover how to create the user interface for an installation.

The Tools Menu

On the Tools pull-down menu, we want to look at just the Customize and the Options commands. When you select the Customize option, the Customize dialog is displayed (Figure 2-6).



The Customize dialog has two tabs: Toolbars and Command. This dialog provides the capability to modify the menu bar and toolbars that are available in the IDE. On the Toolbars tab, there are five built-in toolbars with the Menu bar and the Standard toolbars selected. The MSI Debugger toolbar is applicable when performing debugging in a Basic MSI Project. The Layout and Controls toolbars are applicable when using the Dialog Editor. The operations related to using the Dialog Editor are discussed in Chapter 12.

There are two check boxes to the right of the Toolbars list. When the “Cool look” check box is deselected, all the buttons on the toolbars appear to be three-

dimensional. The “cool look” is to have the buttons with no 3-D effect, the same as found in the later versions of Microsoft's Internet Explorer. The two buttons to the right on the Toolbars tab allow you to create new toolbars that will appear in the list.

On the Command tab, you can add commands to any new toolbar you have created. These commands fall into three categories: those that are already on the Standard toolbar, extra commands that are by default available only from the pull-down menus, and the menus themselves.

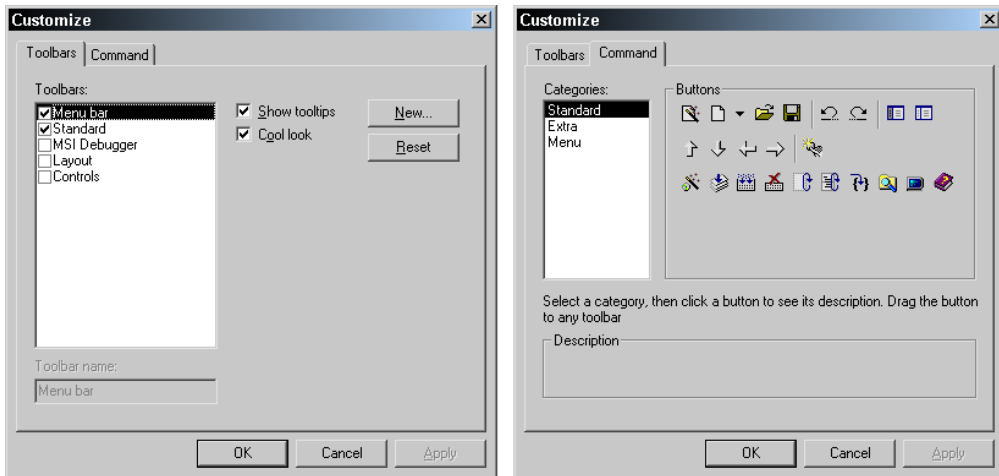


Figure 2-6: *The Customize dialog from the Tools pull down menu.*

If you add extra buttons to one of the default toolbars and then decide to start over or just go back to the default version of the default toolbars, click the Reset button on the Toolbars tab. If you add one or more new toolbars and want to remove them, highlight them on the Toolbars tab and click the Delete button. When you highlight new toolbars that you have created, the Reset button becomes the Delete button. Only the default toolbars can be reset to their original configuration.

Select Options from the Tools pull-down menu to display the Options dialog (Figure 2-7). At this time the item on this dialog in which we are interested is found on the File Locations tab. On this tab, you can change the location where your projects will be built. By default InstallShield Developer sets this location to be in the following location on Windows 2000:

```
C:\Documents and Settings\{Logon Name}\My Documents\My Setups\
```

The following location is defined by the operating system and the `MySetups` folder is created under this location by the InstallShield Developer installation.

```
%USERPROFILE%\My Documents\
```

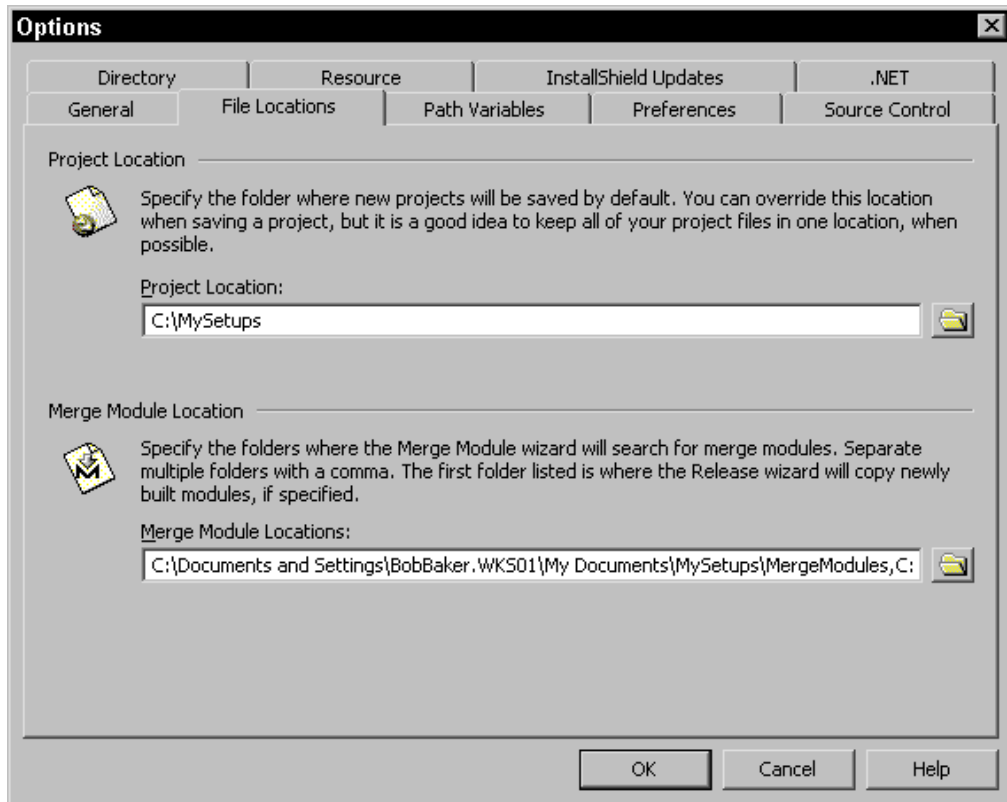


Figure 2-7: *The Options dialog from the Tools pull down menu.*

Throughout this book, you will be frequently navigating to the project build location. Because of this, you should bring this location out to the root of drive C: to make it easier to access. You will be creating all of your projects in the following location:

```
C:\My Setups\
```

To change the project location:

1. Type C:\My Setups\ in the Project Location field.
2. Click OK.

We will be coming back to the Options dialog a number of times to discuss the settings that can be defined. We will do this when discussing operations that these options affect.



On the Help pull-down menu you need to be particularly aware of three commands. The second option on this menu is the online Help for the InstallShield Developer product. The third option on this menu is the help file for the Windows Installer. You will probably use both of these help files to assist you in creating your installation projects. Finally, the fourth

option on this menu is the Readme file for the product.

The Toolbar

The Standard toolbar is shown in Figure 2-8. Like the menu toolbar discussed above this toolbar is also dockable. We can also double click on it and make it into a floating toolbar.

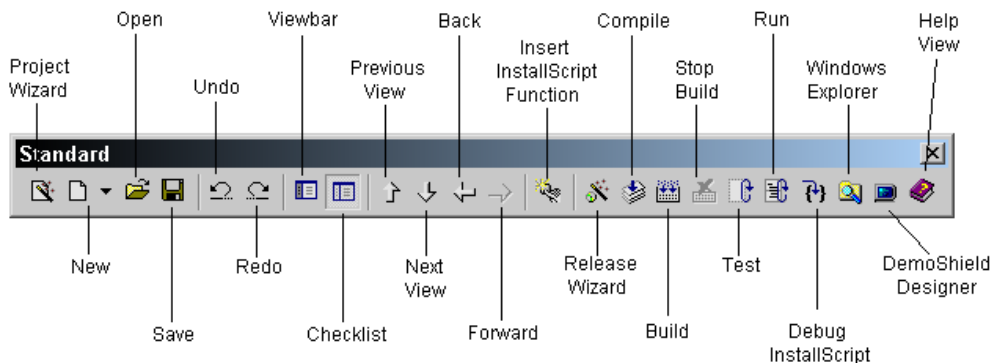



Figure 2-8: *The InstallShield Developer standard toolbar.*

The icons on the toolbar give you an efficient means to perform common operations without having to go to the pull-down menus. If you let the mouse pointer rest on top of an icon, a tooltip is displayed (as long as you did not choose in the Customize dialog to not display tooltips). Now let's look at each of the toolbar icons to see what they do.

Project Wizard: This is another location from which you can launch the Project Wizard. If you already have a project open, clicking on this icon will first ask if you want to save any changes in the open project. After you respond to this dialog, the current project is closed and the Project Wizard is launched. The accelerator key combination for this action is Ctrl+W.



Standard Project...
Basic MSI Project...
Merge Module Project...

New: This icon has a drop-down menu of the three different types of projects that can be created. When you select a project type, the New Project dialog prompts you for a project name and location. The default location is what is set in the File Locations tab on the Options dialog. The default name of the project file is "Your Project Name-X.ism" where X is a sequence number that starts at 1. If you just click on the New icon instead of specifically selecting a project type, a Standard project is created by default.

The accelerator key combinations for creating new projects are provided as follows:

Standard Project: Ctrl+N

Basic MSI Project: Ctrl+B

Merge Module Project: Ctrl+E

Open: This icon launches the Open dialog that takes you to the project file location and allows you to open a project by selecting the desired .ism file. This is the same Open dialog that is displayed if you choose the Open command from the File pull-down menu. If you pull down the "Files of type" combo box in the Open dialog, a list of all the file types that InstallShield Developer can open is displayed. The accelerator key combination to display the Open dialog is Ctrl+O.

Save: Clicking on this icon saves everything in the project, including the project file and any changes that have been made to InstallScript. InstallScript is saved in a separate file and is not in the project file. The accelerator key combination to save your project is Ctrl+S.

Undo: The Undo icon allows you to undo any changes that you made in either the Dialog Editor or the Script Editor. There is a limit of 50 actions that can be undone in the Dialog Editor, but there is an unlimited number of actions that can be undone in the Script Editor. You cannot undo actions anywhere else in a project. The accelerator key combination for undoing actions is Ctrl+Z.

Redo: The Redo icon reverses the Undo action. This applies only to the Dialog Editor and Script Editor. The accelerator key combination for redoing actions is Ctrl+Y.

Viewbar: The Viewbar icon displays an additional vertical bar along the right side of the IDE. The Viewbar provides a different approach to navigating through all the various views in which you work in a project. By default, this view is not active.

Checklist: This icon displays the list of views that serve as a list of the steps that need to be taken to create an installation program or installation package. On the View pull-down menu, this same action is called the View List. The View List is displayed by default.

Previous View: This is a navigation mechanism that allows you to move up the tree of views in the View List. The accelerator key combination for this action is Alt+Up Arrow.

Next View: This is a navigation mechanism that allows you to move down the tree of views in the View List. The accelerator key combination for this action is Alt+Down Arrow.

Back: Clicking this button returns you to the previous view in your history of view selections. The accelerator key combination for this action is Alt+Left Arrow.

Forward: This navigation tool allows you to move forward in your history of view selections. The accelerator key combination for this action is Alt+Right Arrow.

Insert InstallScript Function: This icon is enabled only when the focus is in the Script Editor. It launches the Function Wizard, which facilitates the insertion of built-in functions into a script. This functionality will be discussed when we go over the InstallScript language covered in Part II of this book. The accelerator key combination for this action is Ctrl+I.

Release Wizard: This icon launches a wizard that allows you to define in detail the attributes that are used to build a release of a project. We will discuss this wizard in Chapter 5.

Compile: This icon compiles a script file, links it, and then inserts it into the installation program or installation package. We will learn more about this when we discuss the InstallScript language. The accelerator key combination for this action is Ctrl+F7.

Build: This icon allows you to make a build using the default attributes without using the Release Wizard. It also serves as a means to rebuild a project after you have made changes and do not want to change any of the previous release attributes. The accelerator key for this action is F7.

Stop Build: This icon allows you to stop a build while it is running. This can be handy if you remember something you needed to do before building the installation program or package. The accelerator key combination for this action is Ctrl+Break.

Test: This icon runs the user interface of an installation package, but does not make any changes to the system (for example, it does not copy any files or make any registry entries). The accelerator key combination for this action is Ctrl+T.

Run: This icon runs the installation program or installation package from the IDE. The accelerator key combination for this action is Ctrl+F5.

Debug InstallScript: There is a debugger for script-based programs and this icon runs the installation program in debug mode. The accelerator key for this action is F8.

Windows Explorer: This icon launches Windows Explorer.

DemoShield Designer: This icon launches the InstallShield DemoShield product if it is installed. If DemoShield is not installed, then it displays a message box to inform you that there is an evaluation copy of DemoShield on the InstallShield Developer CD-ROM.

Help View: Clicking on this icon displays the Help view in the IDE, the same as clicking the Help icon in the View List.

Now that you have learned about the InstallShield Developer IDE, it is time to use the Project Wizard. In the next section, you will learn how to create both a Standard project and a Basic MSI project for a small sample application.

The Sample Application

In the remainder of this chapter, you are going to create a Standard installation program and a Basic MSI installation package. Before you can do this, we need to look at the composition of the sample application. Figure 2-9 is a diagram of the features and files that you will use to create the installations.

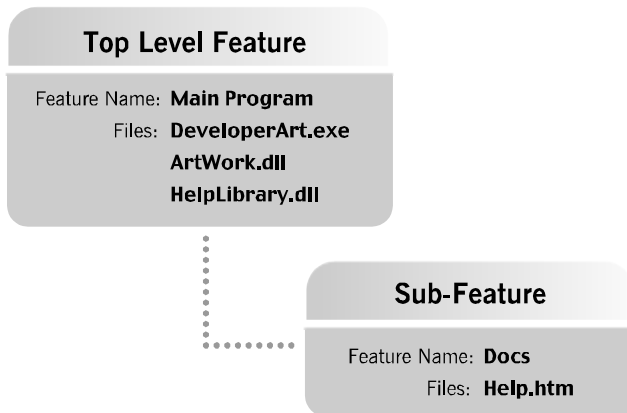


Figure 2-9: *Diagram of the Developer Art sample application.*

This simple application is made up of four files. The Main Program feature is made up of the main executable, a COM DLL, and a Win32 DLL. The file ArtWork.dll is the COM DLL. The Docs sub-feature is made up of a .htm file. Even though this application is small it provides a good introduction to the Project Wizard and its functionality.

Creating a Standard Project Using the Project Wizard

The procedure here will be to step through the Project Wizard panel by panel, discussing each panel in turn and why each action is taken. We will then look at the installation image that is created during the build process. Finally you will install and run the sample application.

It is assumed that you have already set the project location to the following:

```
C:\MySetups
```

In addition, you need to copy the source files for the sample application from the CD-ROM included with this book. The examples in this chapter have the source files located in a folder named as follows:

```
C:\MySetups\Sources\Developer Art
```

There is an installation program on the CD-ROM that will install all the source files, sample projects, and source code to your local machine. The best thing is to use this install program and then to go to the Chapter 02 folder and copy the source files for the Developer Art application to the location specified above. This way you can follow on with the book without any difficulty.

Now, go to the toolbar and launch the Project Wizard by clicking its button. Remember that the Project Wizard can also be launched from the “Create a new project” view and it can also be launched from the Files pull down menu. Click the Next button in the Welcome panel to move to the Wizard Project panel (Figure 2-10). Name this project DeveloperArt_Std so that you can distinguish it from the Basic MSI project you will create. Take note that in this panel you can also select a

previously created project. The difference between the “Open a Recent Project” and “Open an Existing Project” options is that a recent project is one that was created in the project location defined in the Options dialog. An existing project is located somewhere on the system that is not known to InstallShield Developer. That is why this third option allows you to browse to the project file.

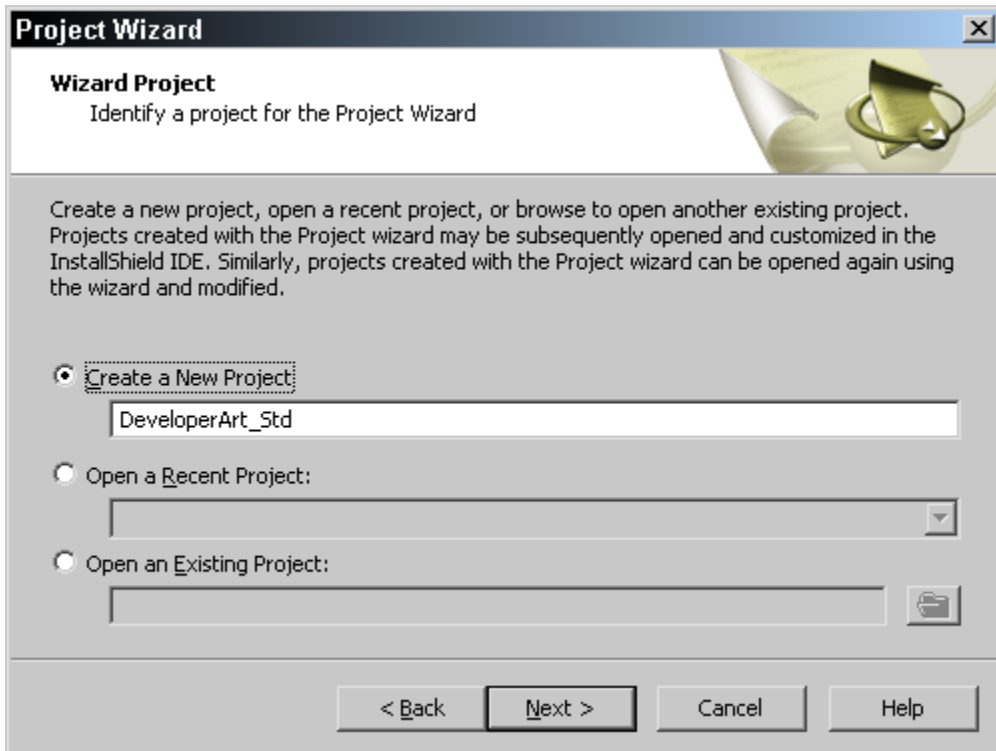


Figure 2-10: *The Wizard Project panel in the Project Wizard.*

Since you are creating a new project, all you need to do is enter the project name and click the Next button to get to the Project Type panel (Figure 2-11). For this first example you are going to create a Standard project. As already stated, this type of project uses InstallScript to interface with the Windows Installer engine.

The main advantage of a Standard project is that you can create a more robust installation user interface than is possible with a pure Windows Installer approach. Other advantages include the use of scripting to perform installation tasks and a smooth upgrade of projects originally created in InstallShield Professional – Standard

Edition. Chapter 12 covers the topic of creating a user interface in both a Standard project and in a Basic MSI project. Chapter 4 discusses some of the details of the run-time architecture for both types of projects. Chapter 11 covers the subject of InstallScript custom actions. Custom actions are the only method for extending the functionality of a Basic MSI project whereas in a Standard project InstallScript can be used to create the user interface as well as extend the functionality of the Windows Installer through custom actions.

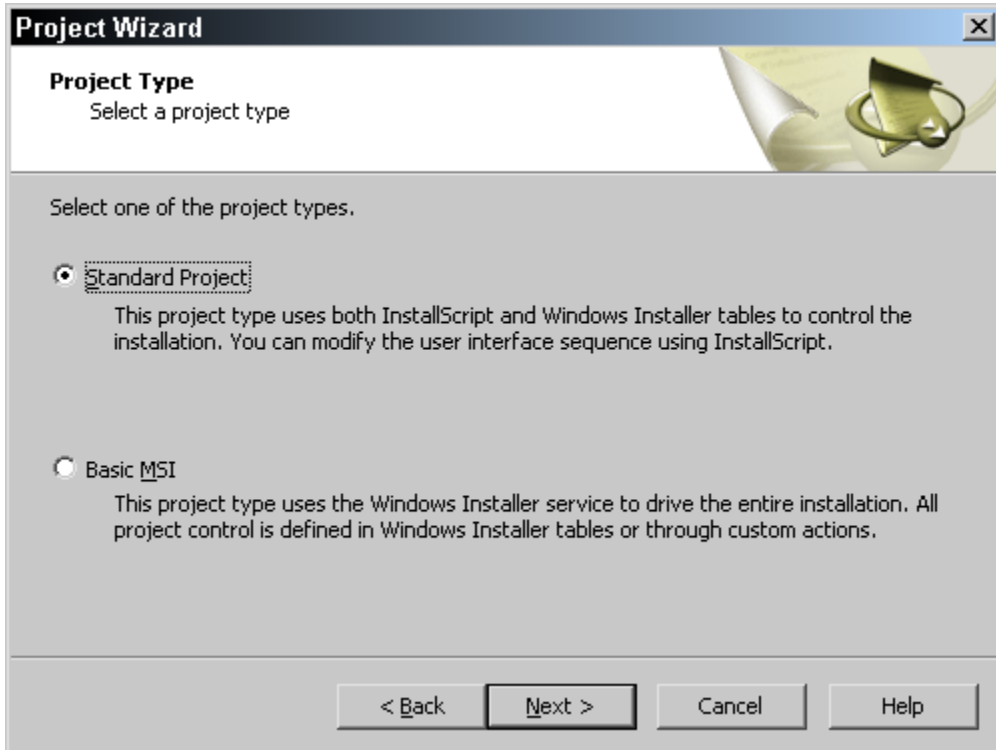


Figure 2-11: *The Project Type panel in the Project Wizard.*

Since a Standard project is the default, click Next to move to the Application Information panel (Figure 2-12). There are three pieces of information that you need to enter in this dialog: the name of the application, its version number, and the default installation location for the application. If you put the cursor in one of the edit fields, a short description of the edit field is displayed at the bottom of the panel. The entries that you need to enter for this project are shown in Figure 2-12.

The entry in the Application Name field can be any descriptive name that uniquely identifies the application to the end user. The value you enter here sets the value of the ProductName property. We will discuss the details of properties and their use in Chapter 3. As you can see in Figure 2-12 the name that is being used for the application includes the project type. This is because you will be creating both types of projects and if they both get installed at the same time then you will not have the problem of mixing the files from one application with the files from the other.

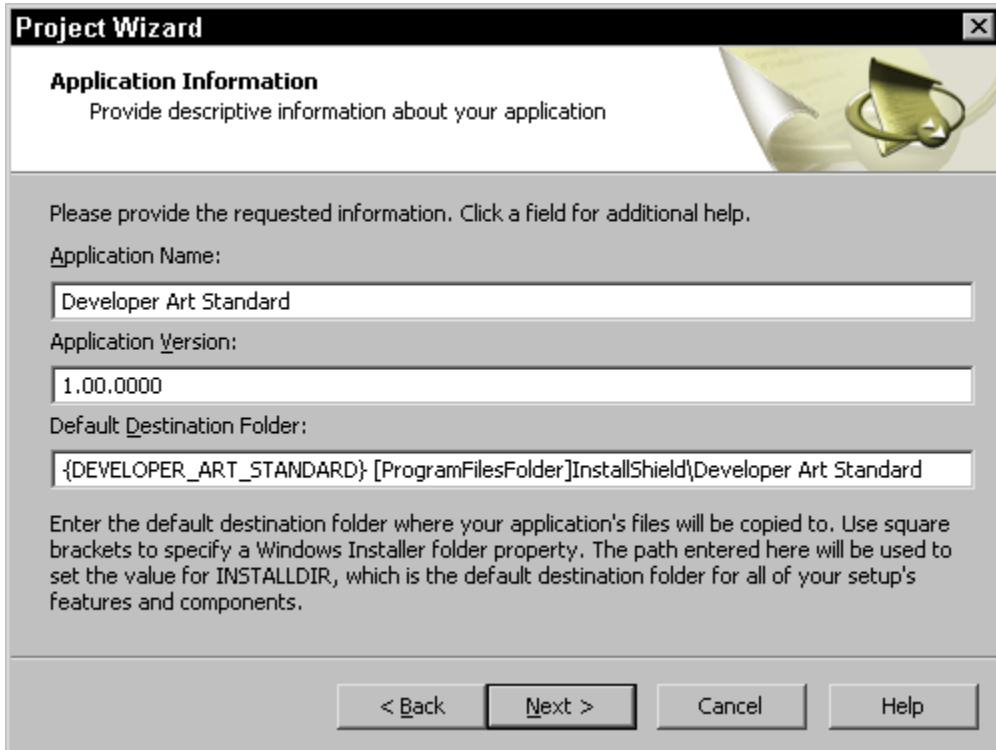


Figure 2-12: *The Application Information dialog in the Project Wizard.*

The Application Version field specifies the version number of the application as a string using a specific format. The format for this string is major.minor.build. The first part is the major version and has a maximum value of 255. The second part is the minor version and also has a maximum value of 255. The third part is the build version or the update version and has a maximum value of 65,535. The value entered for the application version is used to set the value of the ProductVersion property.

The final entry in this dialog is made in the Default Destination Folder field. The entry made here determines the location where all features comprising the application will be installed unless you take specific action to have some features go to a different location. There is an example in Chapter 14 of setting the location of a specific feature to be different than for the application as a whole. The entry that is made here sets the initial value of a property named `INSTALLDIR`. During an installation the location at which this property points can be changed by using a browse functionality that is normally found in a custom setup dialog in an installation's user interface. You might notice some odd formatting in this last field. The string that appears in all capital letters inside the curly braces is called a directory identifier. This identifier is a means to refer to the location that is shown in the Default Destination Folder edit field.

The directory identifier is created from the last part of the path entry we make for the default location. The next entry in the path string is `[ProgramFilesFolder]` and this is an operating system-defined property. This particular property specifies where the Program Files folder is located on the system that the installation targets. The square brackets indicate that the value of this property will be substituted in place of the property and the square brackets. On English systems, this location is usually defined as follows:

```
C:\Program Files
```

Installing to this location by default is a requirement of the "Certified for Windows" logo. Below is a quote from *The Application Specification for Windows 2000 for desktop applications*.

By default, your application must install into an appropriate subdirectory where the current user's program files are stored. This folder is represented by the `ProgramFilesFolder` property in the Windows Installer-based package. (The `ProgramFilesFolder` property is a variable that exposes the path to the Program Files folder, and the Windows Installer sets that variable appropriately on all Windows platforms.) On English systems, this folder is often "C:\Program Files". However, do NOT hardcode that path, as it is not universal.

Exception: If you are upgrading a previously installed version of your application, it is acceptable to default to the directory where that version exists.

The location to which the ProgramFilesFolder property points is found in the registry under the following key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion
```

Before you move to the next dialog, note that there is no backslash separating the [ProgramFilesFolder] part of the default path from the name of the company. This is because all Windows Installer properties that define paths are created with an ending backslash.

In general a name inside curly braces inside an edit field specifies something called a string ID. A string ID is an entry in a string table and this functionality allows us to easily localize an installation to other languages. An example of string IDs inside curly braces can be seen in Figure 2-14.

The use of the curly braces to indicate a directory identifier in the Default Destination Folder edit field shown in Figure 2-12 is an exception to this general rule.

The next dialog in the Project Wizard is the Software Updates dialog. This dialog is shown in Figure 2-13. The InstallShield Update Service is a Web-based service that lets you create self-updating applications with minimal developer effort and no start-up costs. Self-Updating applications allow your end users to easily get updates and information about your product. You author the update using any InstallShield authoring tool, and the Update Service does the rest. The Update Service will notify your end users about the update, download the update from your website and launch the update installation.

In the normal course of creating software you find the need to fix existing defects, add new features, and develop new products and services. One challenge you face is how to deliver these updates to your customers in a timely and efficient manner. It makes sense to distribute them electronically, but without an end-to-end solution that automates customer notification and delivery electronic distribution is problematic. The InstallShield Update Service has been made available to provide such an end-to-end solution.

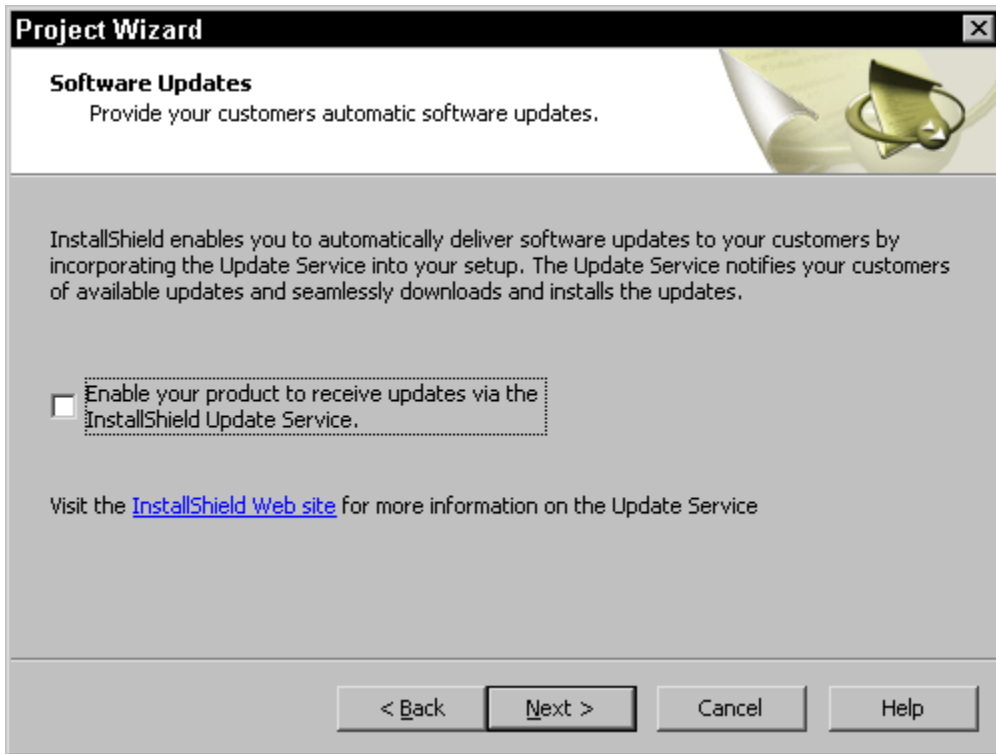


Figure 2-13: *The Software Updates dialog.*

The Software Updates dialog allows you to enable your product to receive updates via the InstallShield Update Service. For this sample application you want to deselect this option in the Software Updates dialog. The Update Service itself is not discussed in this book.

Click the Next button to move to the Company Information dialog (Figure 2-14). There are three entries to be made in this panel. The first entry is the “Company Name” field and this is the name of the organization that developed the application. This entry is used to set the value of the Manufacturer property in the installation program. The second entry is in the “Help Telephone Number” edit field. The entry here should be the telephone number that the end user can use to reach technical support for the application. This entry is used to set the ARPHelpTelephone property and is displayed in the redesigned Add/Remove Programs applet in Windows 2000 and later.

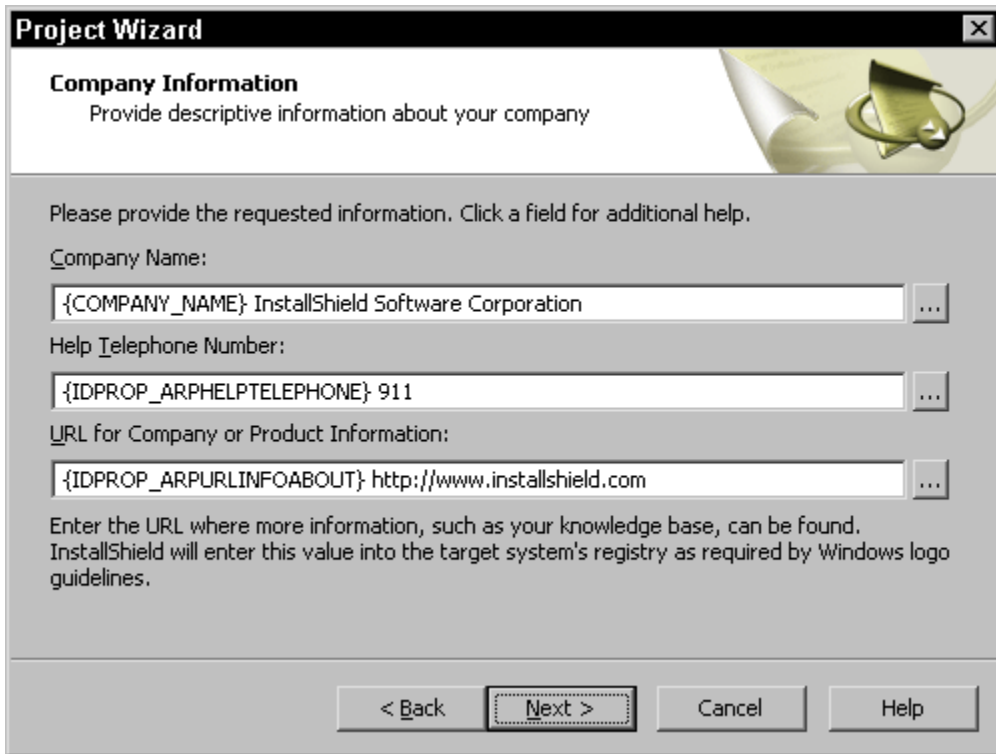


Figure 2-14: *The Company Information panel in the Project Wizard.*

The final entry is in the “URL for Company or Product Information” field. This entry is also made available from the redesigned Add/Remove Programs applet. End users can go directly from the Add/Remove Programs applet to the Web site indicated in this field. The value entered in this field is used to set the ARPURLINFOABOUT property.

You should note several things in this dialog. First is that each of these entries has an associated string ID that is shown inside the curly braces. Also, on the far right of each field there is a small button with an ellipsis. Clicking on this button takes you to the string table where you can select a string that may already be available instead of having to type a new string. If the same string is used multiple locations in a project you should always enter the string by going to the String Table and using the appropriate string ID. This permits you to modify a string in one location only and it will get updated in every location where it is used. This ability to select strings into a text field is available throughout both types of projects.



Figure 2-15: *The Setup Languages panel in the Project Wizard.*

Click Next to move to the Setup Languages panel (Figure 2-15). We need to discuss this panel although you do not need to use it for the installation of the sample application. One of the things that you can do is create an installation program that allows the end user to select the languages in which to perform the install. You can also create a single installation in a language other than English. InstallShield Software Corporation sells two different language packs where all of the default strings have been translated into various languages. These language packs consist of the western European languages and the Asian languages. In this panel, if you try to select a language that is not available, a dialog appears to inform you that the selected language is not available. You then have the option to visit the InstallShield Web site to purchase the necessary language pack.

Since English is selected by default, click Next to move to the Application Features dialog (Figure 2-16).



Figure 2-16: *The Application Features dialog in the Project Wizard.*

It is in this dialog that you define the logical structure of your application by entering the display names of the features that describe this structure. To do this you might want to refer back to Figure 2-9, which shows the structure of the sample application.

This panel provides three default feature names. For this Standard project, delete all of these default features and start over by using the Add button.

1. Delete the existing features by clicking on the feature and clicking Delete. You could also delete a feature by right clicking on it and selecting Delete from the context menu.
2. To add the top-level feature, click on Features and click Add. You can also add a top-level feature by right clicking on Features and selecting New from the context menu.

3. Type “Main Program” as the feature’s name. Press Enter or click elsewhere in the panel to accept the name.
4. Create a sub-feature by clicking on the Main Program feature and clicking Add, or by right clicking on the Main Program feature and selecting New from the context menu.
5. Type “Docs” as the sub-feature’s name.

The names of the features that you have entered here are what will be used as display names in the custom setup dialog. When you create the Basic MSI project, you will use the rename functionality to rename and reorganize the default feature tree. This will give you a chance to work in this dialog using two different approaches for creating features. When you finish the feature tree, it should look like the one in Figure 2-16.

Directly below the list of features is the Destination Folder combo box that is enabled whenever a feature is highlighted in the feature tree. By default each feature has a destination specified as [INSTALLDIR]. Remember that the default value of the INSTALLDIR property is the value we set in the Application Information panel (Figure 2-12). The square brackets, as described earlier, are a replacement mechanism used at installation (run) time. Using [INSTALLDIR] as the feature destination allows the end user to browse for a different location. The selection of a different location changes the value of this property at run time.

Click the down arrow in the right side of the Destination Folder combo box and you will see that you have four possible choices. These choices are shown in Figure 2-17. There are three directory identifiers that can be used to define the default destination location for the features. This is another of those places where the curly braces indicate a directory identifier. The directory identifier {INSTALLSHIELD} is a directory identifier that points to the parent folder of the INSTALLDIR location. Both the {DEVELOPER_ART_STANDARD} and the INSTALLDIR identifiers point at the same location. Because the INSTALLDIR identifier is the typical directory identifier that is used we do not want to select either the {DEVELOPER_ART_STANDARD} or the {INSTALLSHIELD} identifiers. However, either of these directory identifiers defines an acceptable default install location that meets the criteria defined by the “Certified for Windows” logo specification.



Figure 2-17: *The four choices available in the Destination Folder combo box.*

The selection that is interesting and about which we need to talk is the "Browse, create, or modify a directory entry..." option. When you select this option you will display the Browse for Directory dialog box (Figure 2-18).

You can use this dialog to create a directory entry. A directory entry is an identifier that points at a location on the target system. During installation run time, the identifier that you define here becomes a Windows Installer property that identifies a particular path on the target system. We will cover directory entries in more detail in Chapter 3. The directory entries that you create using this dialog must be in all upper-case letters. We will discuss the reason behind this in Chapter 3. The property `INSTALLDIR` is the first example of a directory entry. You use the Browse for Directory dialog to create your own custom directory entries that can be used to direct different features to different locations during an installation. The ability to browse to different locations for different features is functionality that is only

available in a Basic MSI project. Dismiss the Browse for Directory dialog by clicking Cancel.

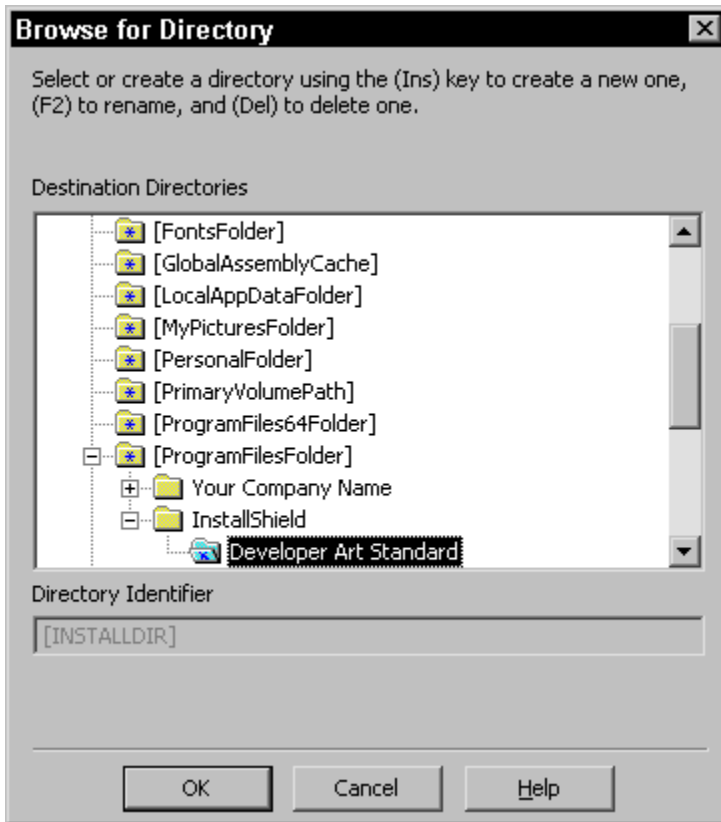


Figure 2-18: *The Browse for Directory dialog in the Project Wizard.*

Click Next to move to the Application Files panel (Figure 2-19). In this panel, you can add files to your application. You add these files to the features that were defined in the Application Features panel. At the top of the Application Files panel is a combo box that lists the features defined in the project. For each feature selected, you can find the application's source files by either browsing for them using the Add Files button or dragging them into the Project Wizard from Windows Explorer.

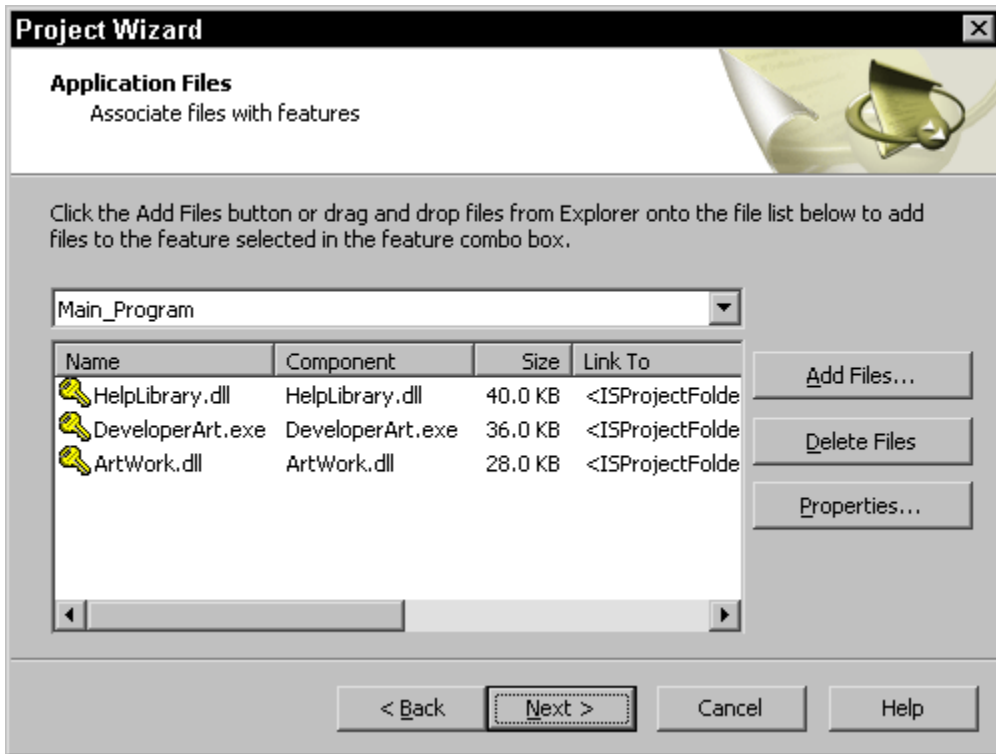


Figure 2-19: *The Application Files dialog in the Project Wizard.*

For this project, you need to add three files to the Main Program feature: DeveloperArt.exe, ArtWork.dll, and HelpLibrary.dll. You also need to add one file to the Docs feature: Help.htm. To add these files use the following steps:

1. Make sure that the Main Program feature is selected in the feature combo box.
2. Click Add Files to display the Open dialog.
3. Browse to the location where the sample project files were saved.
4. Select DeveloperArt.exe, ArtWork.dll, and HelpLibrary.dll and click Open.

5. Now select the Docs feature in the feature combo box and click the Add Files button again.
6. Browse to the source file location and select the Help.htm file and click Open.

After you have added the files to the Main Program and Docs features, the Application Files panel should look like what is shown in Figure 2-19 when the Main Program feature is selected in the combo box. Note the yellow key icons beside each of the files in this figure. When you add files to a feature using the Application Files dialog, components are automatically created in the background. At the lowest level of an installation, the Windows Installer engine is what is used to make changes to the target system. Because of this, even for a script-driven project the installation adds components and not individual files to the target system. The Project Wizard creates the necessary components for you. It is important to understand that to install a file onto the target system, the file has to be associated with a component.

Every component has an attribute called a key path. This key path can be one of four things:

- The name of a file being installed as part of the component.
- A registry entry that is created when the component is installed.
- The folder into which the files in the component are copied during the installation.
- In certain circumstances when installing an ODBC data source the key path of a component can be a reference to this data source.

The Windows Installer writes the key path to the registry at install time as part of the identification of the component on the target system. When the key path is a file and this file has a version it defines the version of the component regardless of how many other files may be installed by the component. In Figure 2-19 the yellow key icon beside the three files shown indicates that these files are the key paths for their respective components. We will discuss the creation of components in more detail in Chapters 3, 5, 13, and 14.

The Application Files panel provides additional information about the files. Scroll to the right to see the following file information: file name, the name of the component created, the file size, the link path to the source file on the build system, the date and time the file was last modified, and the file's installation destination. If you highlight a file and then click the Properties button, the Properties dialog (Figure 2-20) is displayed. This dialog gives you a lot of control over the highlighted file. By default a file will have the same attributes after being installed that it has on the build system. However, you can change this default behavior by deselecting the “Use system attributes” check box and then setting the attributes that you want the file to have after it is installed.

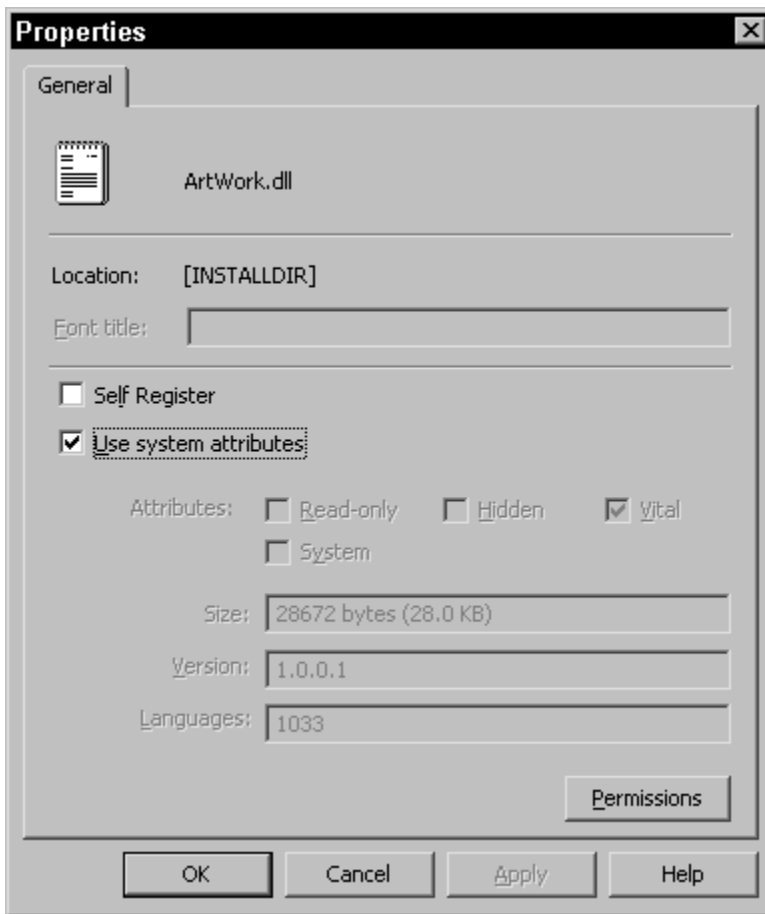


Figure 2-20: *The Properties dialog used to modify file properties.*

There is another attribute called Vital and it is a special attribute used by the Windows Installer when it is copying files to the target system. The file system knows nothing about this attribute and you will not see the letter V beside a file in Windows Explorer. If the Vital attribute is set, and if the component to which the file belongs is selected for installation, then the Windows Installer must be able to install this file for the installation to be completed successfully. If the installer is unable to install the file for some reason, an error dialog box containing “Retry” and “Cancel” options appears. Without this file, the installation cannot continue.

An example of a situation that would cause this to happen is when the source file cannot be located within the source image. It might seem that a file missing from the installation media is an unlikely example but for large applications this is definitely possible. If a file that does not have the Vital attribute set is missing from the installation media, the options presented in the end user dialog are "Abort", "Retry", and "Ignore". The end user can choose to ignore the missing file and successfully complete the installation. Other scenarios that could cause a file not to be found are a dropped network connection or a CD-ROM being ejected from the drive during the transfer of files.

The other parts of the Properties dialog are used for things such as defining companion files and setting font titles for files with a .fon extension. Also you can define a COM server to be self-registered during installation. Self-registration of COM servers in the world of Windows Installer should be avoided; however, it is still supported. We will discuss the issues surrounding self-registration, COM servers, companion files, etc. in Chapters 13 and 14.

On the Properties dialog there is a Permissions button that can be used to secure individual portions of your application in a locked-down environment. This is an advanced subject that is not discussed in this book.

Click Cancel to return to the Application Files panel. Now right click on one of the files displayed in the Application Files dialog and you will get the context menu that looks like what is shown here. The options that are available on this menu are described in the following list:

Add...: Selecting this option allows you to browse to a source location and add files (components) to an existing feature. This is the same functionality as you just used when you clicked the Add Files button.



Copy and Paste: The copy and paste options allow you to copy a file (component) from one feature and add it to another feature. In this fashion you can share components between features.

Delete: Using this option you can remove a component from a feature. Once removed the only way you can retrieve the component is to add the file again using either the Add Files button or the Add option on this menu.

Set Key File and Clear Key File: We have already discussed the business of key paths and these options allow us to set a file to be the key path for a component or to clear the file from being the key path for its component.

Properties: Selecting this option provides the same dialog as is shown in Figure 2-20.

Click Next to move to the Create Shortcuts dialog (Figure 2-21). In this panel, you can define shortcuts in all the usual places such as the Programs Menu and on the Desktop. You can also create non-standard locations where you want shortcuts to be created. To do this you need to go to the Shortcuts node at the top of the tree in Figure 2-21, right click and select the Show Folders option. This option will display a sub-menu of target system locations that are defined by the operating system. If you select one of these locations you will get another entry in the Shortcuts tree under which you can define custom folder names. To create folders under a node in the Shortcuts tree you right click and select the New Folder option. In this fashion you can create as deep a folder tree as you need.

For this application, you will create a shortcut on the Programs menu, which is the standard location to place a shortcut. For the Developer Art application you will need to create a shortcut to the DeveloperArt.exe file that is installed when the Main Program feature is installed.



Figure 2-21: *The Create Shortcuts panel in the Project Wizard.*

You begin the process of creating this shortcut by right clicking on the Programs Menu node and selecting the New Shortcut option. When you do this the Browse for Shortcut Target dialog (Figure 2-22) is launched. What you need to do in this dialog is to browse for the location where the DeveloperArt.exe file is going to be installed on the target system. You use the “Look in” combo box to find this location. When you find the target you want for the shortcut select the appropriate file and click Open.

What you are doing here is selecting the target of the shortcut as defined within your project. You are not actually browsing to the location of the source file on the build machine. In the case of the Developer Art application the DeveloperArt.exe file is going to be installed to the location defined by the INSTALLDIR directory identifier. When you have browsed to the install location for the file that is to be the target of the shortcut you will see the same information about the file as you saw in the Application Files dialog shown in Figure 2-19.

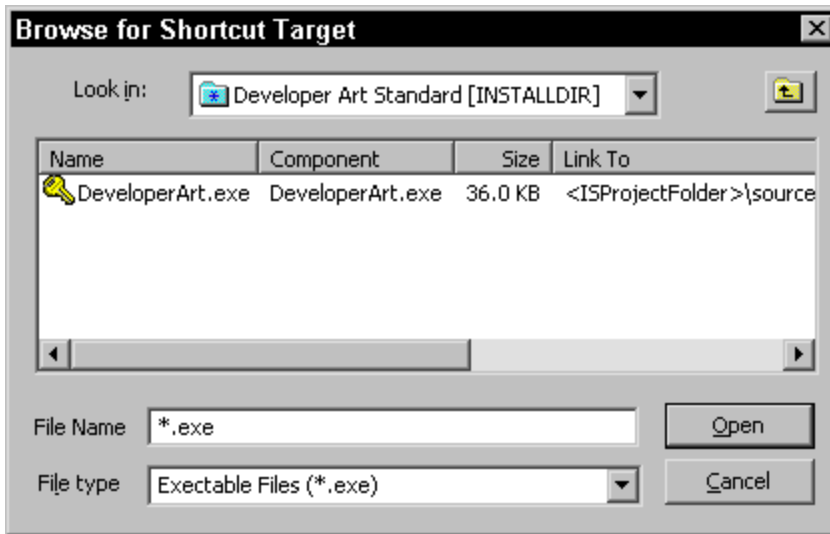


Figure 2-22: *The Browse for Shortcut Target dialog.*

To finish creating the shortcut perform the following two steps:

1. After selecting the DeveloperArt.exe file in the Browse for Shortcut Target dialog enter the string “Developer Art Standard” as the display name of the shortcut to be created. This is done in the Create Shortcuts dialog. What you are defining here is the name of the .lnk file that will be created at the time of installation.
2. Highlight the name of the shortcut just created and then select the feature to which the target of the shortcut belongs. The feature is selected by using the Features combo box. In this case the feature that you need to select is Main Program. You only need to perform this step if the Project Wizard has not already selected the correct feature. When this second step is complete you will have what is shown in Figure 2-23.

Note that in Figure 2-23 the Target combo box contains an entry that defines the target of the shortcut. This will create a standard shortcut and that is the only type of shortcut that can be created using the Project Wizard. In the IDE you can also create something called an MSI shortcut. This type of shortcut is new with Windows Installer and it permits the advertisement of the application. When you advertise an

application with an MSI shortcut on the Programs Menu the application does not get installed until the first time you try to run it. Standard shortcuts cannot be advertised.

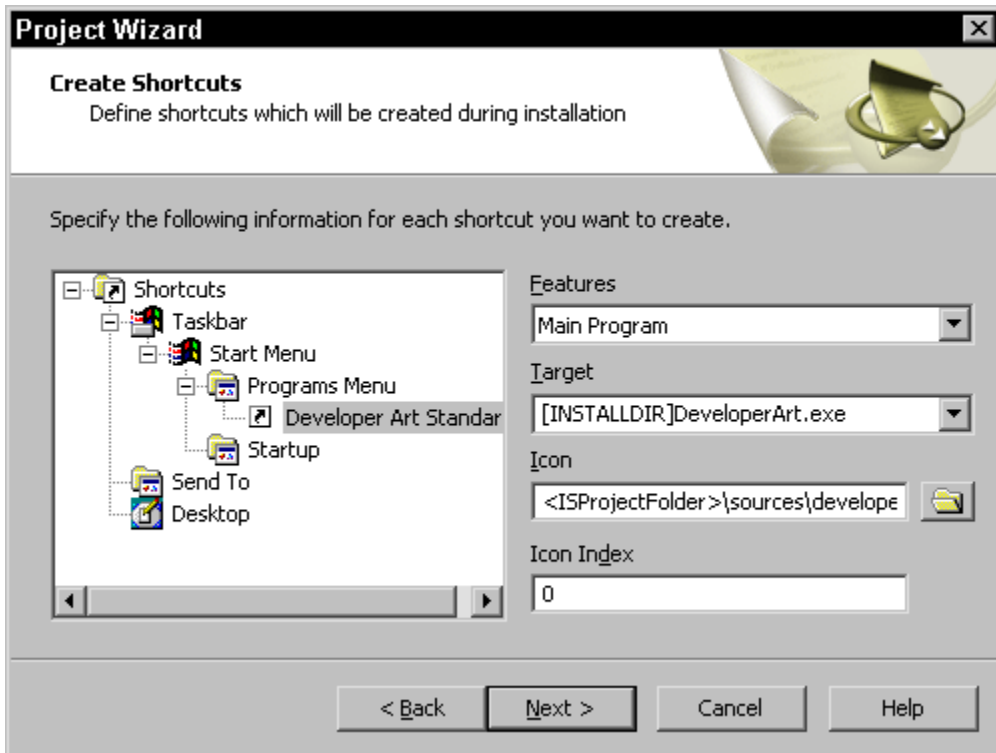


Figure 2-23: *The creation of a standard shortcut.*

Now that you have created a shortcut for your application, you are almost finished creating this installation project. Click Next to move to the Registry Data dialog (Figure 2-24). The Registry Data dialog allows the incorporation of registry entries into the project file by importing the contents of a .reg file. The registry entries are associated with a particular feature and these registry entries are made when that particular feature is installed. What actually happens when you import a .reg file is the Project Wizard will create a new component for you that will be used to make the registry entries at install time.



Figure 2-24: *The Registry Data panel in the Project Wizard.*

Registry files are text files with a special format that have a .reg extension. These files can be merged with the registry on your system to add entries or they can be used, as is the case here, to define registry entries to be made during an installation of an application. REG files can be created using the registry editor or they can be created programmatically. The contents of a simple registry file are shown in Figure 2-25.

```
Windows Registry Editor Version 5.00

[HKEY_CURRENT_USER\Software\InstallShield\Developer]
@=""

[HKEY_CURRENT_USER\Software\InstallShield\Developer\7.0]
@=""
```

Figure 2-25: *The contents of a simple registry file.*

```
[HKEY_CURRENT_USER\Software\InstallShield\Developer\7.0\IDE]
@=""

[HKEY_CURRENT_USER\Software\InstallShield\Developer\7.0\IDE\Workspaces]
@=""
```

Figure 2-25: *Continued.*

For the Developer Art application there is no requirement to import a registry file.

Click Next to move to the Dialogs dialog (Figure 2-26). The Dialogs panel contains a list of the standard dialogs that can be displayed in the installation's user interface. This is the last dialog in which you set any attributes for the installation program.

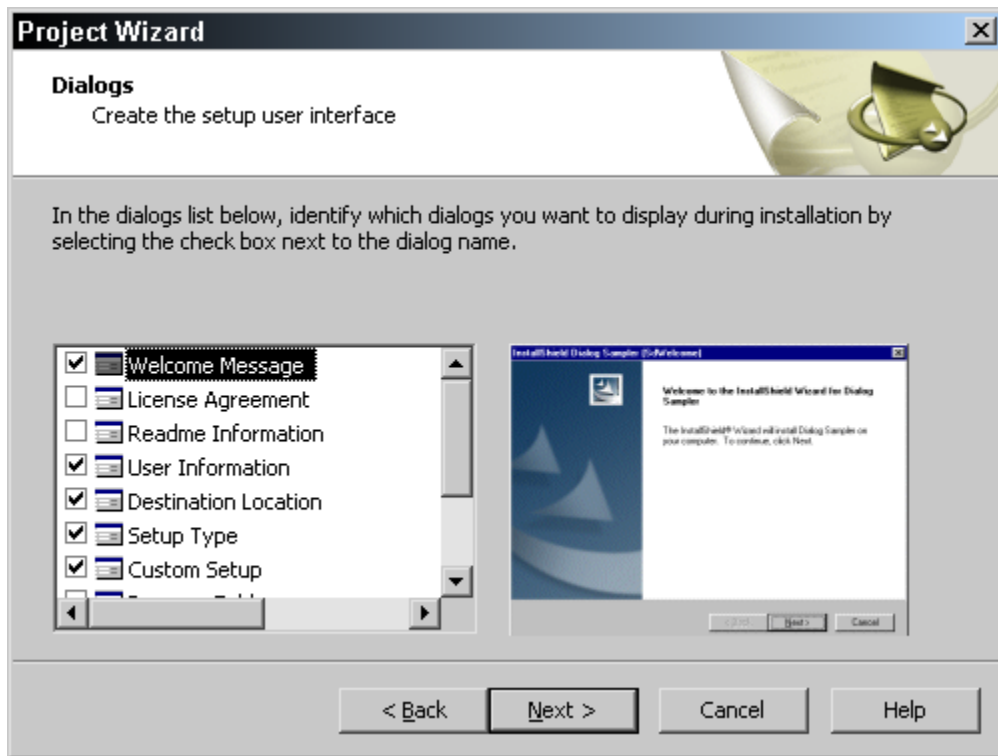


Figure 2-26: *The Dialogs panel in the Project Wizard.*

Highlighting a dialog in the list box provides a preview of the dialog to the right of the dialog list. For this sample application, you do not need to add any dialogs, but

you need to remove one from the user interface. The dialog that you want to remove is the LicenseAgreement dialog. To do this, deselect it the list box.

Now that you have defined the user interface, click Next to move to the Wizard Summary panel (Figure 2-27) where you can review what you have done in the Project Wizard. You can scroll through the summary and see the information that was provided, as well as the setup design of the application. This gives you a chance to go back and fix things before you build the project. You have the option not to make a build if you do not want to build your project. You can prevent a build by deselecting the Build a Release check box at the bottom of this panel. You may want to do this if you know of things that have to be done in the IDE before it is worthwhile to spend the time building. In this case, you do not need to do anything in the IDE at this point. Click Finish to accept the project settings and build your installation program.

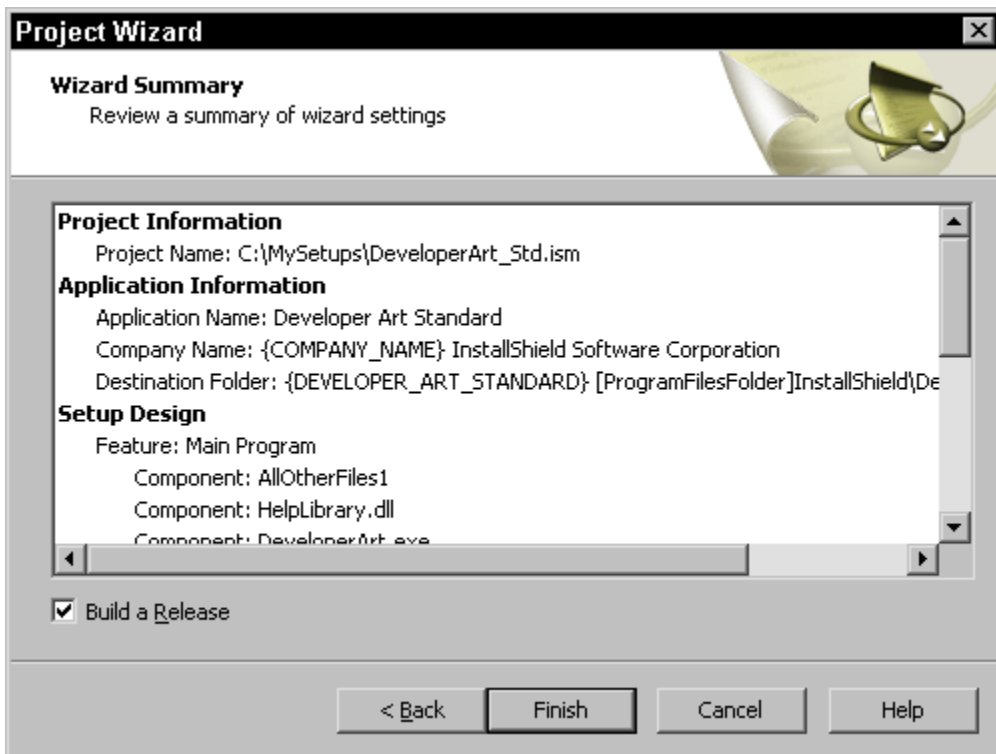


Figure 2-27: *The Wizard Summary panel in the Project Wizard.*

When you initiate the build of your project, the output window (Figure 2-28) opens at the bottom of the screen to provide feedback about how the build is progressing.

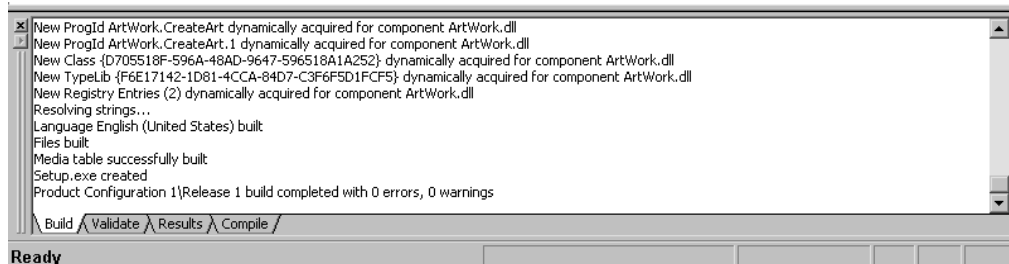


Figure 2-28: *The Output window for the build of the Standard project.*

With such a simple application there should be no build errors generated. You should see a statement at the bottom of the Output window that there were no errors or warnings as shown in Figure 2-28.

Now, you can install the sample application. You can run the installation from the IDE by clicking the Run button on the toolbar. We do not want to spend much time on this now since the user interface is covered in depth in Chapter 12. When you launch the installation the first dialog that you see is the initialization dialog that displays a progress bar that provides the status of the initialization activities. Chapter 4 discusses what initialization activities are being performed.

The first dialog that required action from the user is the Welcome dialog that names the application being installed. When you click Next you move to the Customer Information dialog that provides you with the opportunity to change the default user name and the name of the organization. On Windows NT, Windows 2000, or Windows XP you also have the option of installing the application for just yourself or for all users of the machine.

The next dialog in the user interface wizard is where you are allowed to change the default destination for the application. The destination for the application is defined by the value of the INSTALLDIR property. This property was initialized in the Application Information dialog shown in Figure 2-12. Moving to the next dialog brings you to the Setup Type dialog where you can select the type of installation you want to perform. In this dialog you should select the Custom setup type. When you click Next you will see the dialog that is shown in Figure 2-29. Note that the names of

the features in the feature tree are what you defined when you created the features in the Project Wizard.

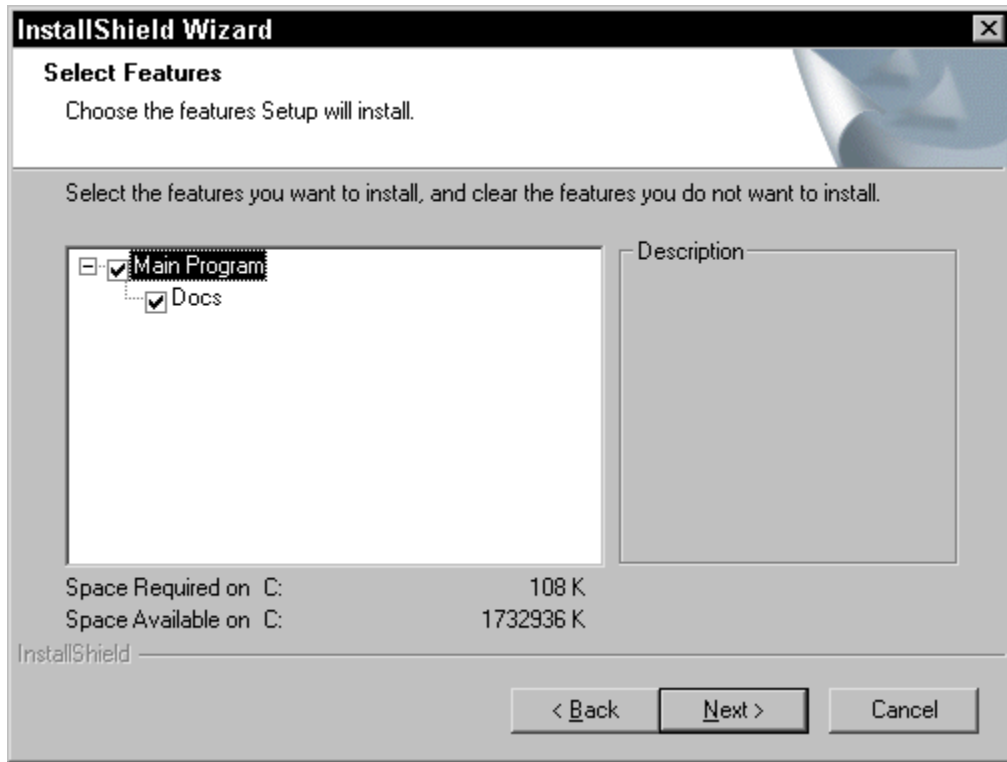


Figure 2-29: *The Select Features dialog for the installation of the Developer Art application.*

The next dialog in the user interface sequence is the Start Copying Files dialog. The purpose of this dialog is to display the settings that were selected during the previous dialogs. Nothing gets displayed in this dialog unless you create InstallScript code to capture the selections. When you click the Next button this will begin the actual changes to the system. While these changes are being made a dialog with a progress bar provides the status of the installation.

Once the installation is complete you should go to the Programs Menu and run the application to see that it works correctly. You will see on this menu the name of the shortcut that you entered in the Create Shortcuts dialog of the Project Wizard.

Now you should try to run the installation a second time by clicking the Run button on the toolbar. After the initialization is complete you will be presented with a completely different dialog than what you had when you ran the installation for the first time. This dialog presents you with the opportunity to perform a maintenance operation on the installed application. In this dialog you want to select the Remove option so that the Developer Art application will be uninstalled.

Creating a Basic MSI Project Using the Project Wizard

Running the Project Wizard to create a Basic MSI project is very similar to what you did to create a Standard project. We will not need to go into detail about those things that are the same. Instead, we will just define the entries that need to be made in the Project Wizard. One of the things in this project that you will do using a different approach is feature creation. This different approach will be discussed in detail.

To begin creating your Basic MSI installation project, launch the Project Wizard and move to the second panel, the Wizard Project panel, where you name your project file. For this project you need to use a different name than used for the Standard project to differentiate between the two. Name this project DeveloperArt_MSI. The .ism file extension is added automatically. Click Next to move to the Project Type panel. In the Project Type dialog, select the Basic MSI option.

Move on to the Application Information dialog and in the three edit fields enter the same type of information that you did when creating the Standard project except that the name of application is different so as to reflect the project type. These entries are shown below:

Application Name: Developer Art Basic

Application Version: 1.00.0000 (this is the default)

Default Destination Folder: [ProgramFilesFolder]InstallShield\Developer Art Basic

In the Software Updates dialog deselect the option to enable the sample application to receive updates. In the Company Information dialog enter the same information that you did for the Standard project with no changes being necessary. These entries are shown below:

Company Name: InstallShield Software Corporation

Help Telephone Number: 911

URL for Company or Product Information: <http://www.installshield.com>

As before, you do not have to take any action in the Setup Languages panel.



Figure 2-30: *The default features in the Application Features dialog of the Project Wizard.*

Click Next to move to the Application Features panel. Here you can use the renaming functionality instead of deleting the default features and creating new ones.

Figure 2-30 shows the Application Features panel as it is first displayed with the names of the default features.

To specify the feature organization for this project:

1. First, delete the DefaultTemplates feature. Select the feature and click Delete, or right-click on this feature and select Delete from the context menu.
2. Rename the DefaultHelpFiles feature to Docs and the DefaultProgram feature to Main Program. To rename a feature, select it and click Rename, or right-click on the feature and select Rename from the context menu. You can also press the F2 to rename the feature.
3. Next, modify the feature tree's organization to make Docs a sub-feature of the Main Program feature. To do this, right-click on the Docs feature and select Move Down from the context menu. This moves the Docs feature below the Main Program feature. Next, right-click on the Docs feature and select Move Right from the context menu. When you complete this set of steps you will have the same feature tree as shown in Figure 2-17.

Click Next to display the Application Files panel. Add the files to the two features just like you did for the Standard project. In the Create Shortcuts Panel, add a shortcut to your project just like you did for the Standard project. The entries here are exactly the same as for the Standard project except that the name of the shortcut should be "Developer Art Basic".

Move on past the Registry Data panel to the Dialogs panel (Figure 2-31). Here you see the first and only major difference, as far as the Project Wizard is concerned, between a Standard project and a Basic MSI project. There is a much smaller selection of default dialogs available. Deselect the LicenseAgreement dialog so that it will not be part of the user interface for the installation of the sample application. After the Project Wizard has created the project, it cannot be used to modify the dialog selection that was set during the initial creation of the project.

You are not limited to only the dialogs listed in the Dialogs panel of the Project Wizard. For a Standard project, you can create custom dialogs using InstallScript and, in a Basic MSI project, you need to make use of the Dialogs view and the Dialog

Editor to create custom dialogs. We will discuss how to create custom dialogs in both project types in Chapter 12.

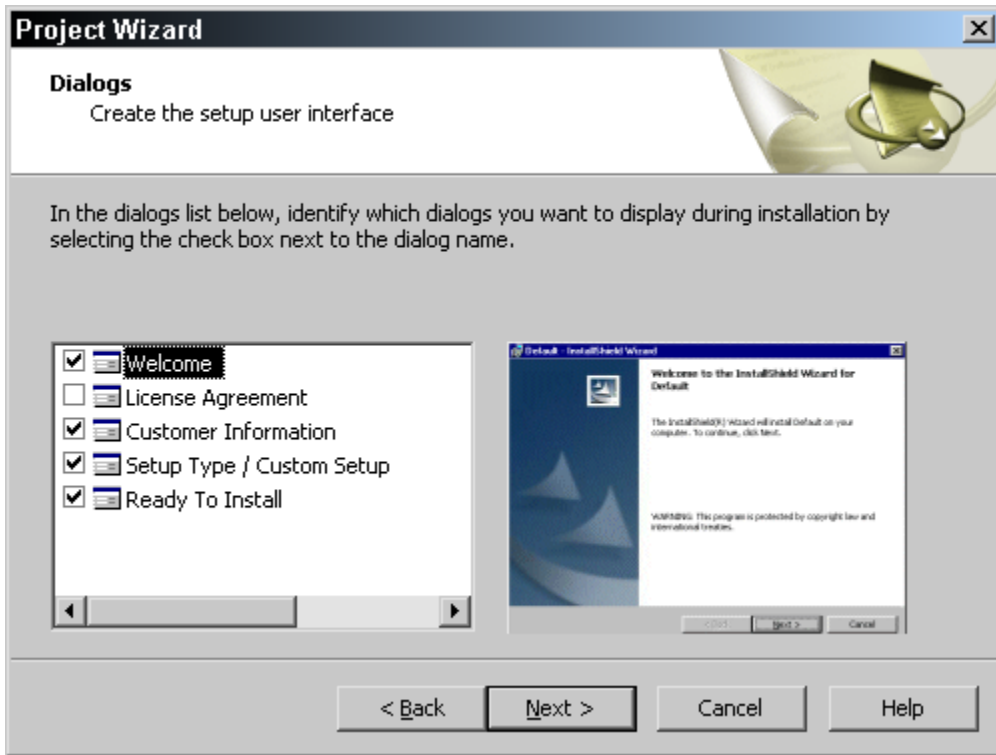


Figure 2-31: *The Dialogs panel in the Project Wizard for a Basic MSI project.*

Click the Next button and move on to the Wizard Summary panel where you can review what you did to create the project. When you are satisfied, click Finish to complete the project and build it into an installation package. At the end of the build you should have the same output window that you had after the completion of the Standard project build except the feedback information will be presented in a different order. The output window should show no errors or warnings.

Now you can run the installation like you did with the Standard installation program. Run the installation by clicking the Run button on the toolbar. When running the installation, accept all of the default settings until you get to the Setup Type panel. You will get a Welcome dialog and a Customer Information dialog just as you did with the Standard project. You will not get a separate dialog for modifying the

installation location for the application. In the default user interface for a Basic MSI project you can only modify the destination location by going to the Custom Setup dialog. When you move to the Setup Type dialog you need to note that the only options presented are to perform a complete or a custom setup. The installation created using the Standard project presented three setup type options.

Select the Custom option on the Setup Type panel and click Next. The Custom Setup dialog (Figure 2-32) displays a different form of the feature tree, one where the feature tree is collapsed. Click on the + to expand the tree.

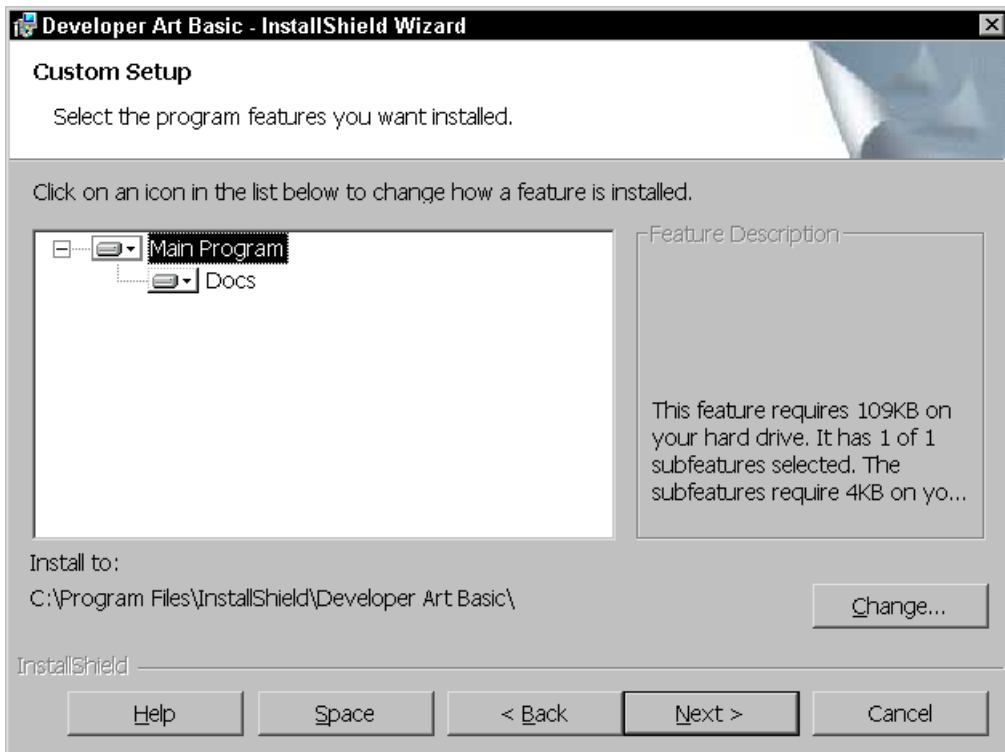


Figure 2-32: *The Custom Setup dialog in the Basic MSI user interface.*

The names of the features are the display names that you entered in the Project Wizard. Click Next to move to the last dialog in user interface sequence. The last dialog before the installation begins to make changes to the system is also different than what you saw with the Standard project. Here there is no list box that can be used to display the selections that were made. There is only a dialog that tells you that

when you click the Install button the installation will begin. After you complete the installation, you should run the application to verify that it works correctly. You will see that the shortcut on the Programs Menu is the same as you entered into the Create Shortcuts dialog in the Project Wizard.

To uninstall the application, use the same approach as used for installation created using the Standard project. Run the installation again, and use the maintenance operation to remove the application. This time you will get a Welcome dialog for the maintenance operation and when you click Next you will get the Maintenance Type dialog where you need to select the Remove option.

Linking to Source Files at Build-Time

Regardless of whether you use the Project Wizard to create your projects or you create them directly in the IDE you will face the problem of finding the files that comprise the application. In a real world scenario the files that make up an application can be located in numerous places, on the build machine, on the network, or some combination thereof. It would be nice if we could have a way to identify these locations without having to continually provide the absolute path to each file that we place in a component. For this purpose InstallShield Developer provides this type of mechanism through something called path variables.

A path variable is a build-time entity that is used in the project to point to the location of the source files that make up the application. You can have many different path variables all pointing at different locations on the build system. An important point to remember is that path variables have nothing to do with where an application is installed on the target system, only where the build can find the files that make up the application.

Normally when you add a file to a component and there is no path variable defined that points at the location where the file resides on the build system, InstallShield Developer will display a dialog asking you to define a path variable for that location. The path variable recommendation dialog is shown in Figure 2-33. This dialog has three possible options that allow you to select between two or more path variables

that point at the same location, to create a new path variable for the location in question, or to specify that the absolute path be used without specifying a path variable.

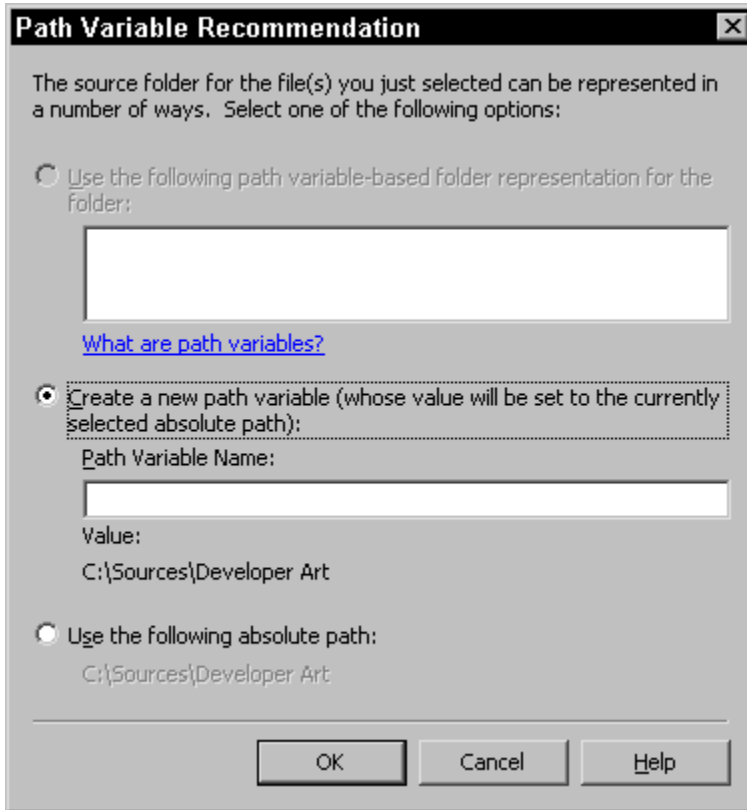


Figure 2-33: *The Path Variable Recommendation dialog.*

The first question you may have in mind is why didn't I see this dialog when the files were added to the Main Program and Docs features in the Project Wizard? The answer is because the source files were located underneath a folder that is pointed at by a predefined path variable. There is a predefined path variable named ISProjectFolder that points at the location where the project is located. In the case of the Developer Art application this predefined variable points at the following location:

C:\MySetups\

Because you placed the source files for the Developer Art application in a folder underneath this location, by default you are not prompted to supply a path variable. If you want to see the Path Variable Recommendation dialog all you need to do is move the source files for the Developer Art application to a location outside of the MySetups folder and then you will get prompted for a path variable.

You have the option to turn off the prompt to supply a path variable and this can be done on the Path Variables tab of the Options dialog as shown in Figure 2-34.

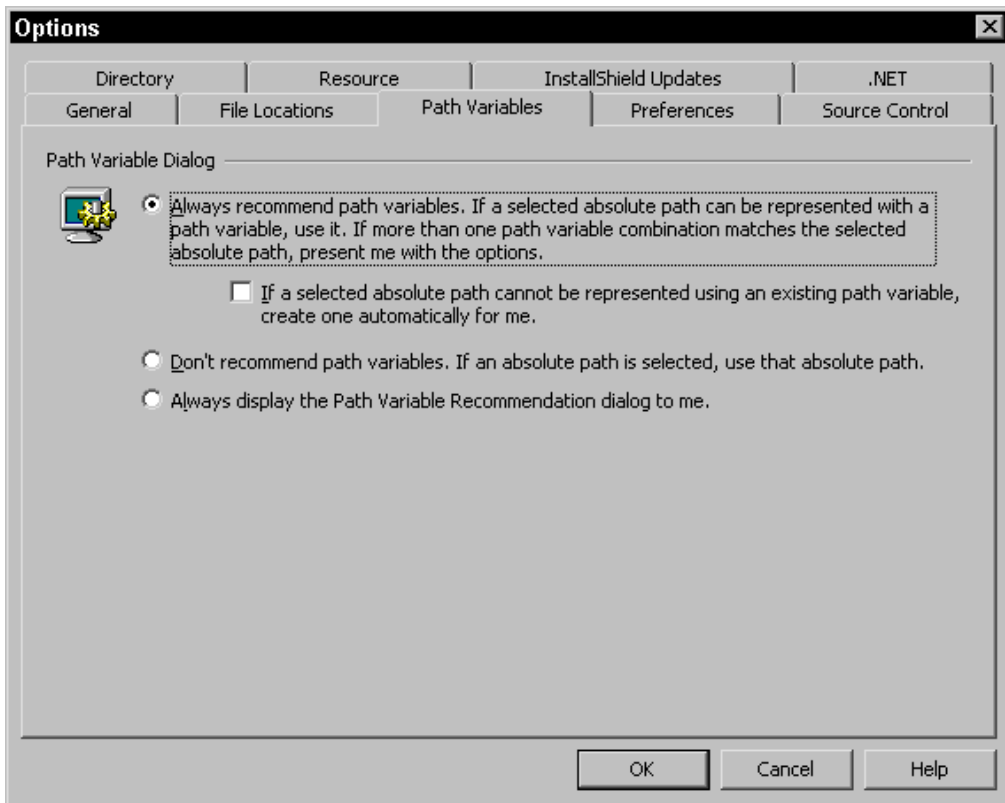


Figure 2-34: *Settings options for path variables.*

As you can see from Figure 2-34 the default is to always recommend a path variable and if there is a path variable that points at a particular location then use this path variable automatically. The other two options in this dialog are to never display the

Path Variable Recommendation dialog or to always display it. The subject of path variables is covered in more detail in Chapter 5.

Conclusion

In this chapter we have discussed InstallShield Developer and how the Project Wizard can assist you in quickly creating a base for your installation project. It is probably evident already that there is a lot to learn here. When we took the quick tour through the product, we reviewed only those items that are important for configuring the IDE. This quick tour, however, gives you a sense of where to find things on the menus and on the toolbar.

You used the Project Wizard to create both types of installation projects: a Standard project and a Basic MSI project. After creating and building the Standard project, you installed the sample application and found that using the Project Wizard created a complete, user-friendly installation program. When you used the Project Wizard to create the Basic MSI project for the sample application, you saw that the entries made were in almost all cases the same as what was used for the Standard project. The only real difference that surfaced in the Project Wizard was the selection of dialogs that could be used for the installation user interface. Even though you did not have to work in the IDE to complete the installations for this simple application it will be a normal thing for more complex applications to have to add additional functionality in the IDE that could not be added using the Project Wizard.

It may appear on the surface that a Standard project and a Basic MSI project are very similar. We will find that under the surface there is considerable difference in how operations are handled during an installation. We will look at these basics in Chapters 3 and 4. The important point to remember is that the Standard project approach uses InstallScript to manipulate the user interface, but uses the Windows Installer to make changes to the target system.

Windows Installer Basics

In the mid-1990s the Office Team at Microsoft decided that they needed to create a new way to install the Microsoft Office suite. They were receiving too many technical support calls related to the installation and uninstallation of Office. Up to this time they had been using another in-house developed installation tool called the Acme Setup Toolkit.

Soon after the Office team began developing a new approach to creating installation programs, it was decided that this new technology should be made part of the operating system so all software vendors could use it. Accordingly the continued development of this new approach to installation development was taken over by the Windows NT 5 team. This is the operating system that became Windows 2000.

Darwin was the code name for this new installation technology. Darwin, later named Windows Installer, became a central piece in the Windows 2000 deployment mechanism. Windows 2000 introduced an entirely new approach to centralized

control over software installation in a networked environment. This centralized software deployment is a major part of Microsoft's Zero Administration Windows (ZAW) initiative, which focuses on reducing the Total Cost of Ownership (TCO) of the traditional desktop PC for corporate America.

This chapter describes the basics of the Windows Installer technology. Understanding this technology is important to being able to use InstallShield Developer effectively because both Standard and Basic MSI projects are founded on this technology. The Chapter 4 describes how InstallShield Developer builds on this technology to enable the two different approaches that it supports.

As you work through the other chapters in this book you will probably want to come back to this chapter and read it again. The content of this chapter will get clearer as you obtain experience in creating installations. The first thing that we want to do is take a look at the issues that guided the design of the Windows Installer.

The Design Focus

The long-term goal of the Windows Installer is to allow the creation of installation packages that require no interaction with the end user and thus take the end user totally out of the equation. Whether this long-term goal will be achieved is unknown, but the first steps toward that goal are fairly impressive. The design of the Windows Installer focused on solving the installation issues of the three major groups that have a vested interest in software installation- the end user, the LAN administrator, and the setup developer. In the early releases of the Windows Installer, the needs of the LAN administrator have received the most attention. However, many of the needs of the LAN administrator are compatible with the needs of the other two groups.

Let's take a look at the design objectives that guided the development of the Windows Installer technology. These are discussed in the following pages.

Be able to install and uninstall software. This may seem obvious, but it is important to note that software installation has been the source of many problems. The capability to install and uninstall software had to be robust in the sense that the target machine would never be left in an unknown state. Many things can happen during an installation; the user can cancel the installation or an error might occur during the installation. The approach taken by the Windows

Installer is to treat an installation as if it were a transaction. The installation happens in its entirety or not at all. For example, if an end user cancels an installation when it is in the middle of copying files, the installation has to roll back the target machine to the state it was in prior to the start of the installation. This means that files and registry entries that are being replaced by the installation need to be cached during installation in order to enable this rollback capability. Because of this, the target machine will always be in a known state.

Be available on all current operating systems. Though this new technology was developed as part of the Windows 2000 operating system, it was necessary to make it available for the operating systems that preceded Windows 2000. These operating systems are Windows 95, Windows 98, and Windows NT 4.0. The Windows Installer is native on Windows 2000, Windows Me, and Windows XP. However, installation programs have to be able to install the Windows Installer engine on those operating systems that do not include it as a native functionality. Furthermore, with the Windows Installer functionality under constant enhancement, it is also necessary to be able to upgrade the Windows Installer on those operating systems where it is a native part.

Be able to control the software installed in a networked environment to only approved applications. What this means is that the desktop can be *locked down*. Generally, administrative or power user privileges are required to install software on an NT-based operating system. This caused problems because giving all users these privileges permitted any user to install software, even if that software might be harmful to the system. If users were not given administrative privileges, then someone with these privileges had to visit each machine individually in order to install or update the required software. This made the corporate network expensive to maintain. The Windows Installer on NT-based operating systems has now enabled a capability that allows the LAN administrator to designate, which users can have specified applications. The designated users do not need to have any special privileges to install approved applications. The LAN administrator is, in effect, granting certain applications elevated privileges on the targeted desktop. On Windows 9.x operating systems, this is not an issue because all users have the equivalent of administrative privileges.

Allow the LAN administrator to know what the installation of a particular installation will do to the target machine. In the past, many installation packages were password protected as well as encrypted. Even installations that

were not encrypted or password had compiled scripts that controlled the installation and these compiled scripts could not be accessed by anyone but the setup developer who created them. These packages were *black boxes* as far as the LAN administrator was concerned. To get away from the black box and allow the LAN administrator to peer inside required that the Windows Installer be an open architecture. Having an open architecture for installation packages means that software vendors cannot use the installation package as the means of protecting their products from piracy. Software vendors now have to implement their protection in the application itself and not in the installation. The main advantage of an open architecture for the LAN administrator is that they can dynamically configure an installation package to filter the features that are available to the end user.

Support the new Windows 2000 deployment capabilities. Windows 2000 deployment makes applications available to the desktop through functionality called advertisement. Advertisement is also supported on Windows 95 and Windows NT 4.0 as long as Internet Explorer 4.01 with service pack 1 is installed along with the Windows Desktop Update. Windows 98 and Windows ME are installed with default support for advertisement. Advertisement makes an application appear to be installed when only registry entries have been made, but no files have been copied. When the end user tries to run the application the first time, the application is installed and then launched. The Windows Installer needed to be designed to support this type of installation mode. This is also referred to as *just-in-time* installation.

Support the concept of resiliency. Resiliency needed to take several forms in the world of the Windows Installer. The Windows Installer needed to be able to gracefully recover from problems during an installation, which is referred to as run-time resource resiliency. It also needed to support the auto-repair of lost components, which is the repair of an installed application when some of its files are deleted by accident. The Windows Installer also had to support the identification of multiple locations for source media. When performing maintenance operations, it is sometimes necessary to have a source for the application files and, if the original source is not available, then it needs to search for other locations. To perform a complete uninstallation of a product the source does not need to be available because the installation database is cached on the target machine.

Provide the end user with a consistent installation experience. This involved having all installations look the same, as if the same company created them all. One of the advantages of the standards used for creating Windows applications is that every application has the same functionality positioned in the same location. For example, most users know that there is an Exit command at the bottom of the File pull-down menu.

Provide a consistent set of rules for creating installations. Providing a set of rules ensures that all installations are created in the same fashion. Consistency in installation creation ensures that installations function in essentially the same fashion. It is also necessary to be able to run a validation on the installation package before shipping the software in order to verify that the rules have been followed.

Be able to manage all shared resources on the target machine. Pre-Windows Installer installation programs had a limited mechanism to track the number of applications that required the same file. Usually, these files were dynamic link libraries (DLLs) that provided generic functionality that many different applications could use. It was important to not uninstall this shared file if other applications on the machine still needed it. The Windows Installer needed to take this functionality much further so that it could know the name of the client for every shared resource. The definition of shared resources now had to encompass all changes to the system and not just the files that were shared. Shared resources now include registry entries and shortcuts, in addition to files.

Support all new operating system features that work to solve problems or enable new programming environments. Each new release of the Windows operating system provides new functionality that works toward solving one or more problems with running applications. Starting with Windows 2000, a new functionality was added that served to protect the core files that make up the operating system from being replaced except in approved ways. Also, starting with Windows 98 SE, a new capability was added that went toward the elimination of file version conflicts, sometimes called "DLL Hell". The Windows Installer needed to support these new capabilities. Also, Windows XP introduced a 64-bit programming environment and the Windows Installer needed to accommodate this as well. Finally, the Windows Installer had to support the new .NET programming environment.

Enable a capability where applications can participate in the management of their environment. To do this, the Windows Installer had to expose an Application Programming Interface (API) so applications could access the functionality of the Windows Installer. This API would allow applications to perform feature level install-on-demand, initiate the installation of other applications, and perform self-repair of its component.

We have now looked at the concepts that drove the design of the Windows Installer and now we need to look at the make up of a Windows Installer package. Remember that both types of installations created by InstallShield Developer - Standard projects, which are script driven, and Basic MSI projects - rest on the Windows Installer. Therefore, both project types create installation programs that consist of a Windows Installer package.

The Windows Installer Package

A Windows Installer package is composed of several elements, with the MSI file as the central element. We call it the MSI file because it has an .msi extension, which stands for Microsoft Installer. A conceptual diagram of the contents of an MSI file is shown in Figure 3-1.

The standard elements contained in all MSI files are a set of database tables and a Summary Information Stream. There are also two optional elements shown in the diagram, with the first of these being a cabinet file that contains the files of the application in a compressed format. If the files that make up the application are not compressed into a cabinet file, they are external to the MSI file. In this case, information in the database points to where these files are located on the distribution media. It is also possible to have cabinet files that are external to the MSI file. The other optional element that is included inside the file as a sub-storage object is a transform. People who are familiar with COM will recognize that an MSI file is a COM Structured Storage file. We will discuss what this means in the next section.

A Windows Installer package defines the installation for one and only one product. You might wonder how one creates an installation for a suite of products such as Microsoft Office. The answer is that the suite is considered the product and the various applications that make up the suite are features of the suite. A product is composed of features. Features are part of the logical structure of an application and

represent the end user's view of the application's functionality. Features are composed of components and components are what provide the features with their functionality. Components contain the installable parts of an application, which can be files, folders, registry entries, and shortcuts. A component can be thought of as the application developer's view of the product. End users never see components.

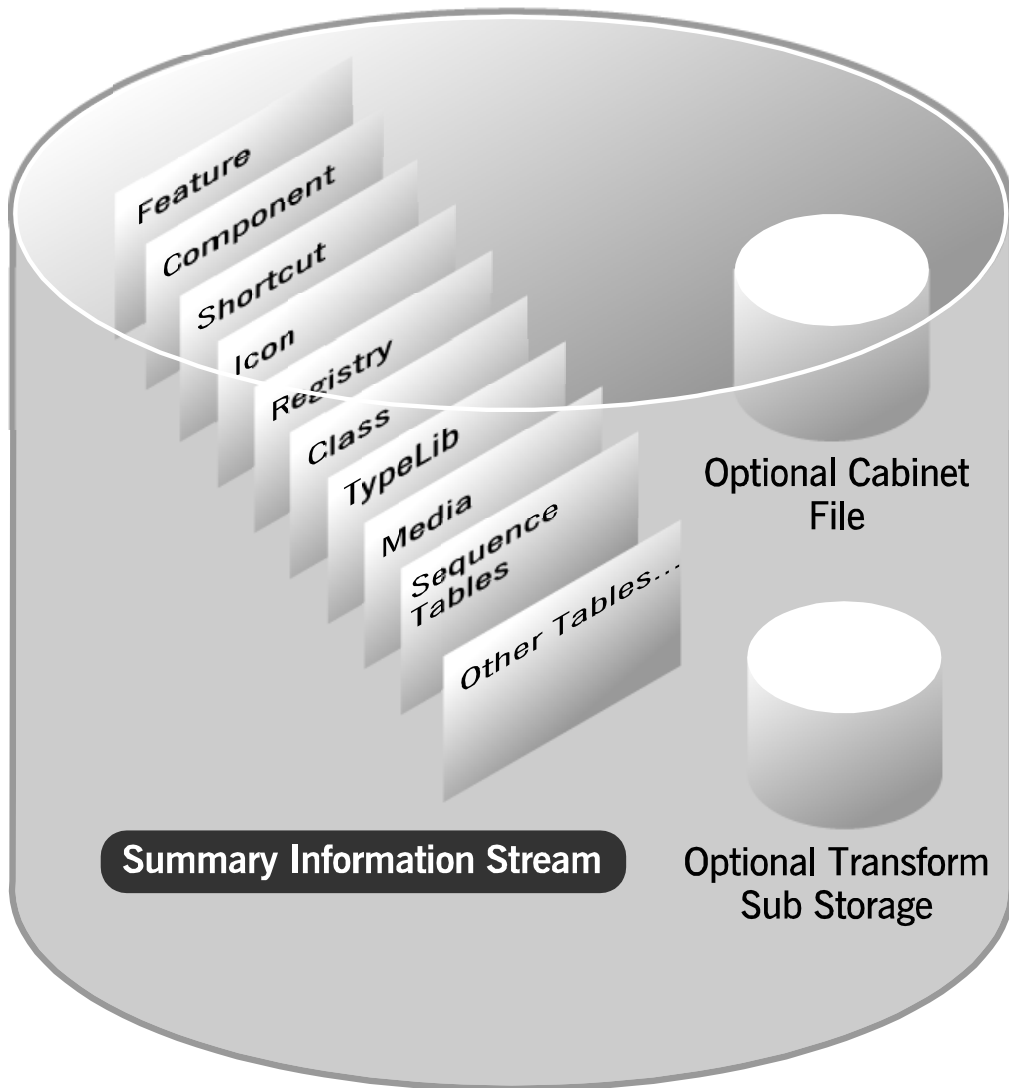


Figure 3-1: *Diagram of the elements of an MSI file.*

Before we get into more detail about the elements of an MSI file, we need to first discuss the design of a COM Structured Storage file. It is important to understand the concepts of this file type since we use terminology that comes from this technology.

What is a COM Structured Storage File?

A COM Structured Storage file, sometimes called a compound file, is a file system within a file. This type of file contains two distinct types of entities, storages and streams. A diagram of a sample COM Structured Storage file is shown in Figure 3-2.

We can see in Figure 3-2 that at the top of the file there is a storage called the root storage. Under the root storage there are either sub-storages or streams. A sub-storage can have either other sub-storages or streams. It is easy to see the analogy between a file system and a COM Structured Storage file. The root storage is analogous to the root directory in a file system. The sub-storages are analogous to the sub-directories and streams are analogous to files.

The concept of COM Structured Storage files was created as one of the persistence mechanisms for applications using COM. Persistence is a term used to describe the saving of data in some permanent fashion, such as in a database or a file. This type of file allows two different applications to share the same file to store their data. Using this type of file is how a Microsoft Word document can contain the text of a document, a graphic object, and an Excel spreadsheet. This file type was originally called a docfile because Microsoft Word pioneered the use of this type of data storage in the early days of COM (called OLE in those days). Another benefit of using this file type is that an application does not have to save the complete file, but only that part on which it has made changes. One of the disadvantages of a COM Structured Storage file is that this type of file is bigger than a normal flat file that would contain the same information all in one format.

The function of a storage object is to keep track of the storages and streams that are below it. A storage object can contain any number of other storages and streams just like sub-directories in a file system. A storage object does not contain any data, it contains streams and the streams contain data. A stream is a container for bytes and it is the application that created the stream that organizes this stream of bytes. COM does not organize the stream of bytes. A stream is always viewed as a contiguous byte array, however the bytes themselves do not have to be contiguous within the COM Structured Storage file. This further enhances the analogy between a stream and a file.

As with a file, only the application that creates the stream is able to interpret the contents of the byte array that composes the stream.

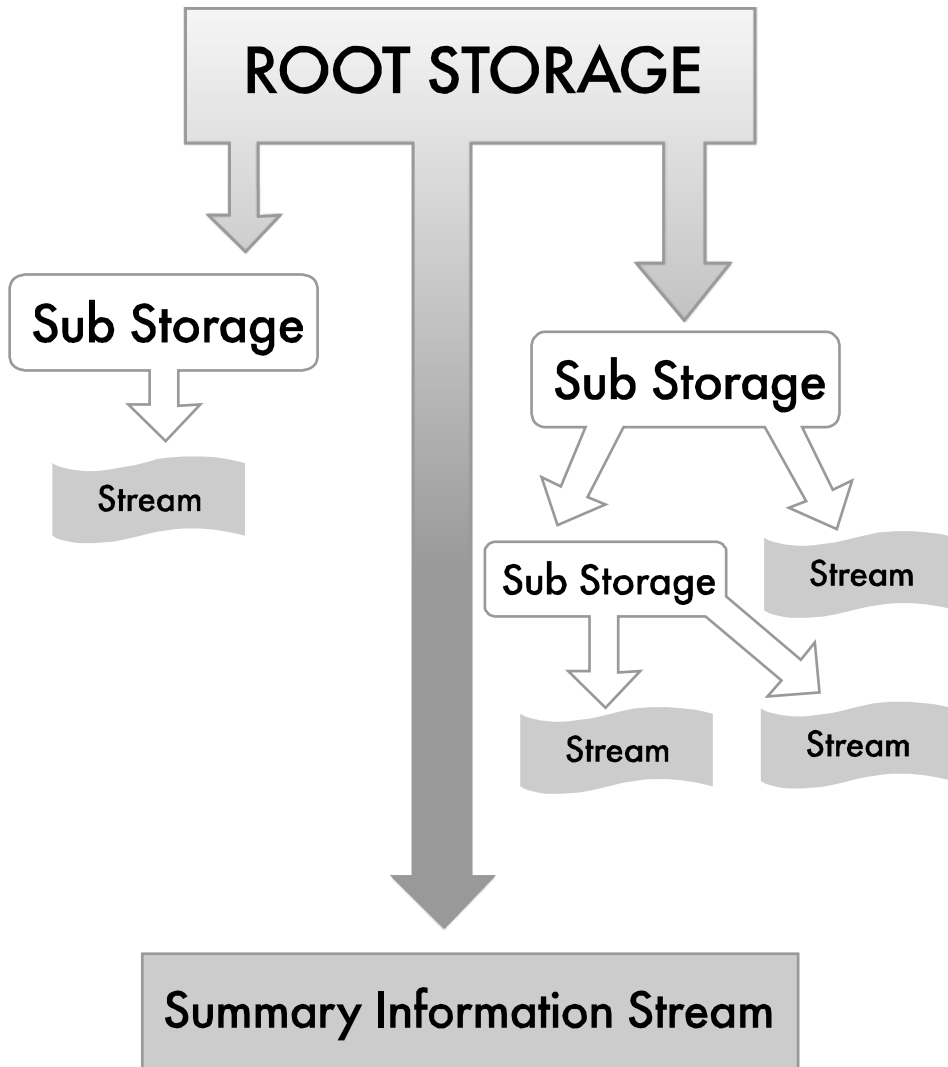


Figure 3-2: *A COM Structured Storage file.*

Each storage and stream object in a COM Structured Storage file must have a name. In general, these names are not exposed to any user so the names can be anything that is useful to the file creator. However, just as in a file system, there are certain

rules that need to be followed when naming storage or stream objects. Except for the root storage, the names of storage and stream objects are limited to a maximum length of 31 characters. The name of the root storage is the name of the COM Structured Storage file and its naming is defined by the allowable names in the file system. The other restriction on the naming of storage and stream objects is that these names cannot start with any character that has an ASCII value of 0x00 to 0x31 inclusive. Also storage and stream object names cannot include a backslash (\), a forward slash (/), a colon (:), or an exclamation point (!). The names of storage and stream objects are case sensitive.

In Figure 3-1, we see a stack of rectangles each with a different name. This represents the tables in the database contained in an MSI file. Each table in the database is a stream and the name of the stream is the name of the table. Figure 3-1 indicates that there can optionally be a cabinet file included inside the MSI file. The Windows Installer supports multiple cabinet file streams in an MSI file. InstallShield Developer, however, will create only one cabinet file stream when it builds an MSI package. When a cabinet file is included inside the MSI file, it is included as a stream. The name of the cabinet file stream must follow the rules for the naming of streams as discussed above. When a cabinet file is external to the MSI file, it needs to follow the short file name convention of the file system. The other optional element inside of an MSI file is a transform sub-storage object. This is the only type of sub-storage object that is supported in an MSI file. Transforms are briefly discussed at the end of this chapter.

In both Figure 3-1 and Figure 3-2, there is a stream called the Summary Information Stream that is stored directly below the root storage object. In an MSI file, the inclusion of a Summary Information Stream is necessary. For normal COM usage the creation of a Summary Information Stream in a structured storage file created by an application is highly recommended but not required. A Summary Information Stream is an implementation of a COM entity called a property set. A property set is a means of storing information in such a way that any conforming application can manipulate the information in the property set.

The Summary Information Stream is the most commonly used property set. To see this, open Windows Explorer and right-click on an MSI file or a Word document and select the Properties option. In the Properties dialog, click on the Summary tab to see the contents of the file's Summary Information Stream. Figure 3-3 shows the Summary tab on the Properties dialog for the MSI file generated by the Basic MSI project that you created in Chapter 2. In the Windows Installer, the Summary

Information Stream is more than a way for users to view a set of properties in an MSI file. The values in the Summary Information property set are critical elements in defining the Windows Installer functionality for any installation package.

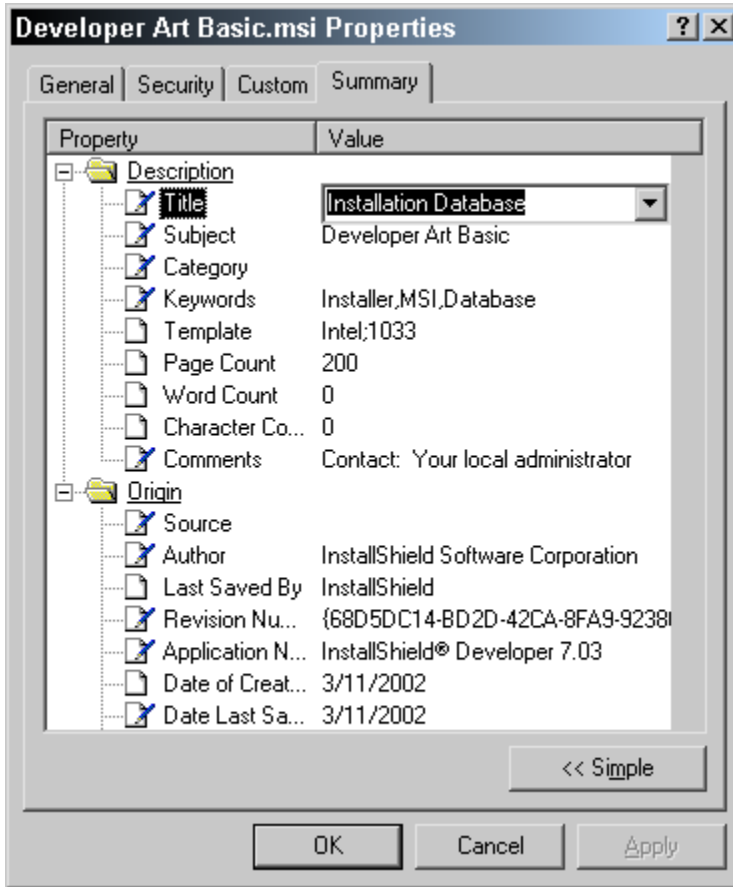


Figure 3-3: *Windows Explorer properties dialog.*

The name of the Summary Information Stream in a COM Structured Storage file is the string "\\005SummaryInformation". This is the name for this stream as defined by the OLE2 specification. The leading character is one of those that we are not allowed to use. The character '\\005' indicates an element name that is for the exclusive use of COM and other system code built on it such as OLE Documents. There is another Summary Information property set defined by Microsoft Office that has a very similar name thus leading to confusion. The name of this Summary Information

stream is "\005DocumentSummaryInformation". The Windows Installer does not use this type of Summary Information stream. The whole purpose of this second type of Summary Information property set is to allow users of Microsoft Office to create custom properties.

Like an MSI file, the InstallShield Developer project file uses structured storage. Now it is time to take a closer look at the various elements of an MSI file.

The Summary Information Stream

In the previous section, we discussed the function of the Summary Information Stream in a COM Structured Storage file. In addition to displaying information about the MSI file in Windows Explorer, the property values are used by the Windows Installer to install the application defined in the database tables. It is important to understand that the Summary Information Stream is not considered one of the database tables. When we talk about the installation database, the Summary Information Stream is not included. It is a separate entity.

The property set that composes the Summary Information Stream in an MSI file consists of 17 properties. Four of these properties are required and must have values. Note that in some cases, the name of the property has no relationship to the use made of the property's value. We will look at these properties later in this chapter when we dissect the MSI package that you created for the Developer Art application in Chapter 2.

This same property set is used for the Summary Information Stream in the other types of Windows Installer packages that can be created. However, some of the properties have different uses depending on the type of package that is created. The names of the properties are no different than those used in the MSI file. The other types of Windows Installer files that also contain a Summary Information Stream are merge modules, transforms, and patch packages. Each of these Windows Installer file types is discussed at the end of this chapter.

The Windows Installer Database

As discussed, the database tables in the MSI file are all streams in the MSI file. Currently, there are 84 permanent tables and 7 temporary tables defined in the

Windows Installer version 2.0 database-schema. The term schema as it is used here means the design of the database as defined by the table names, column names, permitted column data, the designated primary keys, and the foreign keys. The temporary tables in the database are only for implementing the building of the installation package and are not shipped with the database on the distribution media.

It is in the database tables that you provide the information that the Windows Installer requires to perform the installation. Very few applications need to use all the tables in this database. The installations that you will develop in this book use only a small percentage of the tables defined in the schema.

The design of the Windows Installer database uses the relational model. This means that the various tables interact with each other through the columns that are designated as foreign keys. It also means that each table has one or more columns that are designated as the primary key for the table since a foreign key has to identify a primary key in the table with which it is interacting. By definition, a primary key must uniquely define a row in a table. Because of this, duplicate primary keys are not permitted.

The relational model also requires data integrity. The three types of data integrity that we are most interested in as setup developers is referential integrity, field-level integrity, and internal consistency.

Referential integrity: Referential integrity is a state in which all foreign key values in a database are valid. For a foreign key to be valid, it must contain either the value NULL, or an existing key value from the primary or unique key columns referenced by the foreign key. In other words if a foreign key references the primary key of another table, then that primary key must exist for there to be referential integrity.

Field-level integrity: Field-level integrity is a state where the values in all fields are of the proper data type and conform to the domain of values that are valid for a particular column. In the Windows Installer database there are only three types of data recognized; strings, integers (two-byte and four-byte), and binary streams. However, an installation package requires specific integers or strings in certain columns of a table. The specification for each column of each table is maintained in the `_Validation` table. For example, the `FileName` column of the `File` table is a column that takes a string data type, but it specifically stores the name of a file. Therefore, not only should your entry be a string, but it should also follow the

requirements for naming files. Field-level integrity also requires that columns that are not allowed to be null contain valid values.

Internal consistency: Internal consistency relates to how the data in various columns interact with each other. It is possible that each individual field in every table in the database is valid when examined from the perspective of field-level integrity, but that they are invalid because they cause incorrect behavior of the database as a whole. For example, the Component table might list several components that are all valid when evaluated individually from a field-level integrity perspective. However, evaluating only the Component table would not catch the error when two components use the same GUID as their component code. To find these types of errors, you need to evaluate the databases that you create from an internal consistency perspective.

The Windows Installer permits a lot of flexibility in working with the installation database. Setup developers can create custom tables and InstallShield Developer does exactly that in many areas. It is also possible to add temporary rows and columns to tables in the database at install time. However, to work with the database either at build time or install time, you need to have knowledge of SQL. There is only one table where the Windows Installer provides special functions so that the use of SQL is not necessary. The table where SQL is not needed to get or set values is the Property table. The Windows Installer has its own special version of the SQL query language.

Compressed and Uncompressed Source Files

When you create an installation package, you can have compressed source files, uncompressed files, or some combination of compressed and uncompressed files. All uncompressed source files must exist in a single source tree and the root of this tree is defined in the Directory table in the database. Normally the root of the source tree is defined by the location of the MSI file on the distribution media.

When source files are compressed, the compressed files must be included in a cabinet file. A cabinet is a single file, usually with a .cab extension, that stores compressed files in a file library. The cabinet format is an efficient way to package multiple files

because compression is performed across file boundaries, which significantly improves the compression ratio.

InstallShield Developer uses the cabinet file creation tool `Makecab.exe` to generate cabinet files for use with the Windows Installer packages that it builds. `Makecab.exe` has three key features:

- Storing multiple files in a single cabinet file.
- Performing compression across file boundaries.
- Permitting files to span cabinets. Large files can be split between two or more cabinet files. There can be no more than 15 files in any one cabinet file that spans to the next cabinet file. For example, if you have three cabinet files the first cabinet can have 15 files that span to the second cabinet file and the second cabinet file can have 15 files that span to the third cabinet file.

`Makecab.exe` supports three levels of compression: None, MSZIP, and LZX. MSZIP is the default compression type supported by Microsoft. The LZX compression method can achieve higher compressions ratios. Using MSZIP compression generates a cabinet file approximately the same size as a PKZIP-compatible compression engine. LZX compression requires more time, but LZX decompression is typically faster. InstallShield Developer supports the MSZIP and the LZX compression algorithms, but does not create a cabinet file that has uncompressed files in it.

A cabinet file can be located inside or outside of the MSI file. To conserve disk space, the installer always removes any cabinets that are embedded in the MSI file before caching the installation package on the end user's computer. If a cabinet file is not embedded in the MSI file, then this external cabinet file must be located at the root of the source tree which is normally the location where the MSI file is located.

The installer extracts files from a cabinet as they are needed by the installation and installs them in the same order in which they are stored in the cabinet file. The space requirements for installing a file stored in a cabinet are no different than for installing an uncompressed file.

The Windows Installer also supports cabinet files created with the `Cabarc.exe` creation tool. This tool writes to the Diamond cabinet structure, but it is considered

only a basic cabinet creation tool and does not have the capability of the Makecab.exe tool.

The Windows Installer SDK

Before we examine the Windows Installer package that you created in Chapter 2, you need to download the Windows Installer SDK. You will use this tool as you go through this book. To install the Windows Installer SDK, do the following:

1. Go to the following Microsoft URL:

<http://www.microsoft.com/msdownload/platformsdk/sdkupdate/>

2. Click on the Windows Installer SDK link on the left of the page.
3. Click on the Install this SDK link on the right hand side of the page.

Note that it is necessary to first install the Core Platform SDK before installing the Windows Installer SDK. The default location of the installation will be under Program Files in a folder called MsiIntel.SDK. Navigate to this location and expand the folder.

For now, the only thing that you need to install is the database-editing tool that comes with the Windows Installer SDK. The name of this database-editing tool is Orca and the installation package for it is in the Tools folder under the MsiIntel.SDK folder. Double-click on the package to install Orca. Select the complete installation setup type.

Orca.exe is a database table editor for creating and editing Windows Installer packages and merge modules. The tool provides a graphical interface for validation, highlighting the particular entries where validation errors or warnings occur. You will use this tool extensively to examine the various aspects of the Windows Installer technology.

Now it is time to take a close look at the Windows Installer package that you created with the Project Wizard in Chapter 2. Here we will go into the rudimentary details of the installation package.

Dissecting the Developer Art Installation Package

This section examines the three major elements of the Basic MSI installation package created in Chapter 2; the Summary Information Stream, the MSI database, and the distribution image. You will become familiar with the Orca database-editing tool as you use it to look at the package you created.

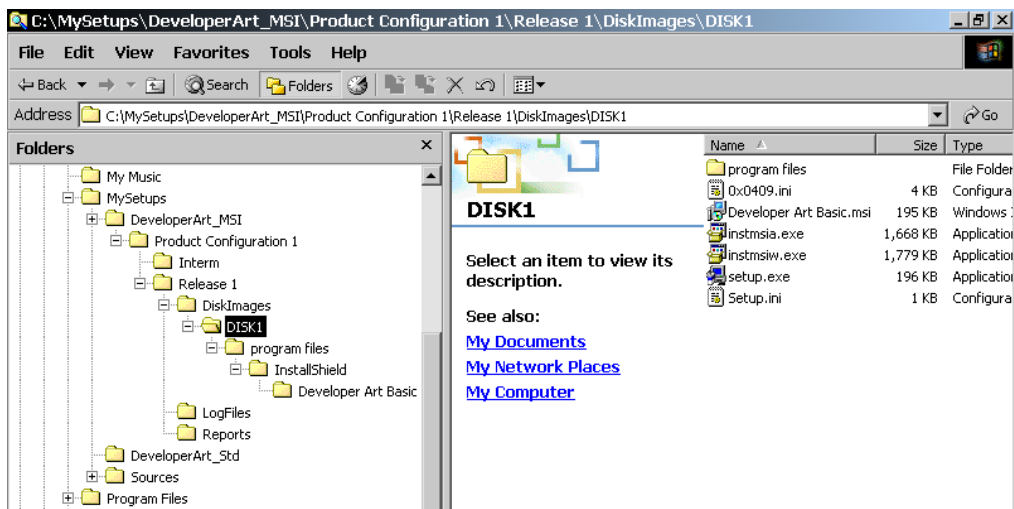


Figure 3-4: *The build location for the Basic MSI installation package for the Developer Art application.*

First, navigate to the location where the installation package was created. This location is shown in Figure 3-4.

This location under the MySetups folder is the default build tree the Project Wizard created for you. Everything under the DISK1 folder is part of the distribution image. If you wanted to distribute this application on a CD-ROM, you would copy everything under the DISK1 folder to the CD-ROM. Now, however, we are interested only in the file with the .msi extension. You will learn about some of the other files in Chapter 4.

The Developer Art Summary Information Stream

To view the values that were placed in the Summary Information Stream property set for the Developer Art Basic.msi MSI file, go to the DISK1 folder as shown in Figure 3-4 and right-click on the file. Select the Properties option and click on the Summary tab in the dialog that appears.

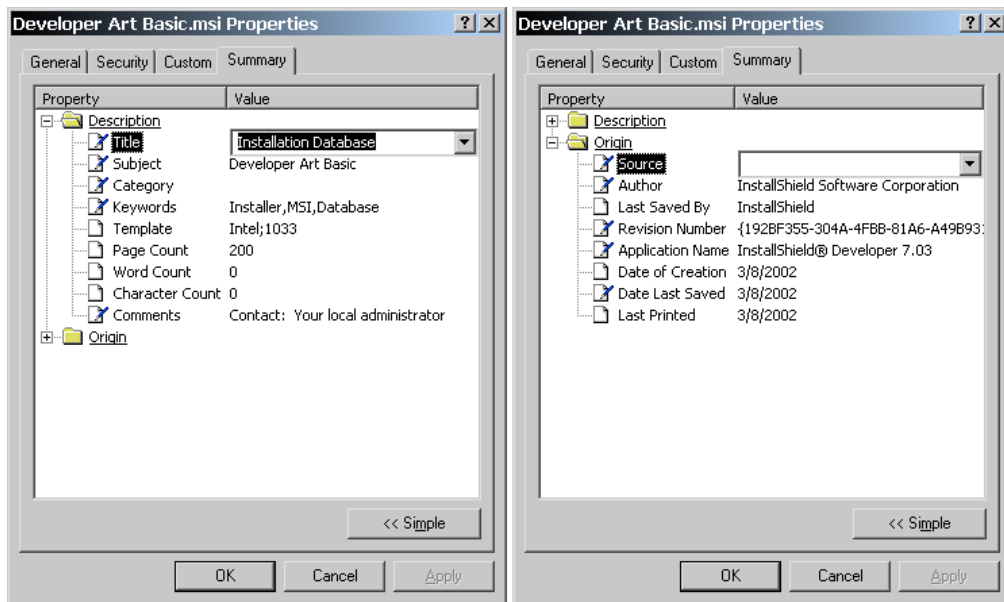


Figure 3-5: The Summary Information Stream property values for Developer Art Basic.msi.

Ensure that the advanced view is showing. You should see what is shown in Figure 3-5. Computers not running the Windows 2000 operating system will display a different Properties dialog than the one shown in Figure 3-5. The following section covers the properties shown in Figure 3-5 and discusses what the values means and why they are there. The names of the four required properties are underlined.

Title: The Title property briefly describes the type of installer package. Microsoft recommends such phrases as "Installation Database" or "Transform" or "Patch" for the value of this property. As shown, InstallShield Developer uses the recommended string from Microsoft. The only purpose that this property serves

is to inform anyone who is unfamiliar with the .msi extension the purpose of this file type.

Subject: This summary property identifies to the Windows Explorer the product that can be installed using the logic and data in the Windows Installer installation package. This value was set from the Application Name entry you made in the Application Information panel in the Project Wizard.

Category: The Category property is not part of the Windows Installer Summary Information Stream property set. That is why the value for this property is blank.

Keywords: Windows Explorer uses this property when someone wants to perform a keyword search on the system for a special type of file. The default set of keywords used by InstallShield Developer is shown in Figure 3-5 but it can be modified to add such things as the name of the application. It is up to the setup developer what they want for the value of this property. Note that a comma is used to separate different keywords.

Template: The Template property indicates the platform and language versions supported by the MSI database. This is one of the four required property values that need to be set. The syntax of the this property information is:

```
[platform property][,platform property][,...];[language id][,language id][,...].
```

The valid values for the platform property are Alpha, Intel, and Intel64. Note that the Alpha platform is not supported by Windows Installer version 1.1 or later. If the platform of the target machine does not match one of the platforms specified, the Windows Installer will not process the installation package. If a platform is not specified, then this implies that the package is platform-independent.

Entering 0 in the language ID field of this property, or leaving this field empty, indicates that the package is language neutral. By default, InstallShield Developer makes the string "Intel;1033" the value of this property. This means that it will install only on an Intel-based machine and that the language supported by the MSI database is English.

Page Count: For an installation package, the Page Count property contains the minimum installer version required. This is one of the four required property

values. For example, to specify that Windows Installer version 1.0 is acceptable; this property must be set to the integer value of 100. For 64-bit Windows Installer Packages, this property must be set to the integer 200 since earlier versions do not support 64-bit Windows. The formula for determining the correct value for this property is to multiply the major version by 100 and then add the minor version without the decimal point. Remember that the only type of numerical value supported by the Windows Installer is either 2-byte or 4-byte integers.

Word Count: This property is a bit field consisting of three bits. The purpose of this property is to indicate the type of source file image. This is one of the four required property values. The possible values for this property are given below:

Value	Meaning
0	The source files are uncompressed and long file names are used.
1	The source files are uncompressed and only short file names are used.
2	The source files are compressed and long file names are used.
3	The source files are compressed and only short file names are used.
4	The source files are in an administrative image and long file names are used.
5	The source files are in an administrative image and only short file names are used.

For the Developer Art application the value is set to 0. This means that you are using an uncompressed set of source files and that this application can be installed only on a system that supports long file naming. If an installation package had a combination of compressed and uncompressed files, this value

would be set to 0 and the File table would mark which files were compressed. By definition, an administrative image is uncompressed.

Character Count: This property is not used in MSI files so it is Null for the Developer Art installation package.

Comments: This property conveys the general purpose of the installer database. By convention, the value for this summary property is set to "This installer database contains the logic and data required to install <product name>." <product name> is the name of the product being installed. In general, the value for this summary property changes only in the product name, nothing else. InstallShield Developer puts in a default string, which you need to change in the IDE in order to conform to this convention.

Source: This is not a property that is in the Windows Installer Summary Information Stream property set.

Author: The value of this property is the name of the company that created the application that is being installed by the MSI package. This value is set from the Company Name field of the Company Information panel in the Project Wizard.

Last Saved By: The Windows Installer sets this property to the name of the user that is logged on to the system during an administrative installation. The installer never uses this property and a user never needs to modify it. Setup developers can use this property to track the last person to modify the database. This property should be left set to Null in a final shipping database. InstallShield Developer uses the value of the RegisteredOrganization registry entry to set this property.

Revision Number: This property contains the value of the package code for the Windows Installer package. This is one of the four required property values. The package code is a globally unique identifier (GUID) that uniquely distinguishes this particular package from any others. A GUID is a unique 128-bit number that is guaranteed to be unique and never duplicated anywhere else in the world. This is the same type of identifier that COM uses for class IDs and many other purposes. This identifier is used in the standard registry format, which consists of a number of sequences of hexadecimal values separated by dashes (-). The final form of this identifier is to surround the hexadecimal values with curly braces

{}). We will see that the GUID is used in many places in the Windows Installer to make things unique.

Application Name: The name of this property in the Windows Installer Summary Information Stream is "Creating Application". This property identifies the application that created the Windows Installer installation package. In general the value for this summary property is the name of the software used to author the database.

Date of Creation: The name of this property in the Windows Installer Summary Information Stream is "Create Time/Date". The value of this property is the date on which the installation package was created.

Date Last Saved: The name of this property in the Windows Installer Summary Information Stream is "Last Save Time/Date". This property conveys the last time the installer database was modified. Each time a user changes an installation, the value for this summary property is updated to the current system time/date at the time the installer database was rebuilt.

Last Printed: This property can be set to the date and time during an administrative installation to record when the administrative image was created. For non-administrative installations this property is the same as the "Create Time/Date" property.

These are the Windows Installer Summary Information Stream properties that are displayed on Windows 2000 in the Windows Explorer properties dialog. There are two additional properties that are not displayed.

Codepage: This property is the numeric value of the ANSI code page used for any strings that are stored in the Summary Information Stream. This does not have to be the same code page for strings in the installation database. The Codepage property is used to translate the strings in the Summary Information Stream into Unicode when calling the Unicode API functions. This property must be set before any string properties are set in the Summary Information Stream.

Security: This property identifies whether the package should be opened as read-only. A database-editing tool should not modify a read-only enforced database and should issue a warning at attempts to modify a read-only recommended

database. The following values of this property are applicable to Windows Installer files.

Value	Meaning
0	No restriction.
2	Read-only recommended.
4	Read-only enforced.

You can also view some of the Summary Information Stream property values using Orca by doing the following:

1. Right-click on the Developer Art Basic.msi file and select Edit with Orca, to open the MSI database in Orca.
2. From the View pull-down menu, select the Summary Information option. The Summary Information option displays several of the properties, although not all of the properties that are shown in Figure 3-5.

Next, we will look at the database tables that are required to describe the operations that are necessary to install the Developer Art application.

The Database Tables

As you look at the Developer Art Basic.msi file in Orca, note that there are 30 permanent tables and 1 temporary table in the database that contain one or more rows. You can distinguish permanent from temporary tables by the fact that temporary tables have a leading underscore () as part of their name. In this section, we will look at each of these tables and see what information is in them and how that information is used during an installation. To make this investigation easier, we can group these 31 tables in seven categories. Because this is a relational database, some tables appear in more than one category. The definitions of these seven categories follow:

Application Design Tables: The tables in this category are used to define the feature and component structure of the application. Remember that features are the end user's view of the application as seen in a custom setup dialog and components are the developer's view of the application. Components contain the files that are copied to the target system.

File Copy Tables: These tables define the information that is necessary to move the required files from the distribution media to the target machine. Considering that you might have compressed and uncompressed files, as well as files that are not copied but run from the distribution media, this is not as simple as it may appear on the surface.

Registry Entry Tables: These tables define the registry entries that are required during the installation. Most installations make numerous entries for COM, uninstallation information, product information, and other items. The registry is considered the replacement for the initialization files that used to be used by applications.

Installation Procedure Tables: There is a set of tables that is used to control the operations carried out during an installation. The tables in this category instruct the Windows Installer when to perform a particular operation.

User Interface Tables: Most installation programs need to provide a user interface. The user interface leads the person performing the installation through the steps required to initialize the environment so the installation can be completed successfully. The tables in this category are used to design the user interface necessary for a particular product.

Desktop Integration Tables: One of the final things that most installation programs need to do is to expose the application to the end user. This is normally done through the creation of shortcuts at various locations in the operating system shell, file associations, and MIME types. The tables in this category define this operation.

Installation Validation: The one temporary table mentioned above is used to perform a validation on the database. It is always a good idea to run a validation on the installation database before shipping a product. In fact, it is required to pass Microsoft's validation suite as the first step in meeting the requirements for obtaining the "Certified for Windows" logo.

We have discussed the definition of the various categories of database tables needed for the installation of the Developer Art application. We now need to get into the details of each of these categories. Before we do, however, we need to discuss the syntax used in the database schema diagrams that are used to discuss the various categories of tables.

Database Schema and Relationship Diagrams

The syntax that is used to illustrate the schema of the tables in the database and the relationship between tables is shown in Figure 3-6.

Figure 3-6 depicts that a table will be shown with its name at the top with the names of the columns that make up the primary key shown in a rectangle at the top. Below the rectangle that contains the primary key are the remaining columns in the table. Beside each column name, the type of data that goes into the column is indicated. To the right of the data type is an indicator of whether a Null entry for the column is acceptable.

We will not go into a lot of detail about the different data types that appear in the database schema diagrams. Just remember that, at the lowest level, there are only three basic data types: strings, integers, and binary streams. You can look at the Windows Installer help file for definitions of the various data types. You can access this help file using the InstallShield Developer Help pull-down menu or with the Windows Installer SDK under the Help folder.

One data type in particular is very common. This is the Identifier data type that is used in many tables as the primary key for the table. This data type is a text string with certain restrictions. Identifiers may contain the ASCII characters A-Z (a-z), digits, underscores (), or periods (.). However, every identifier must begin with either a letter or an underscore and it cannot contain any spaces.

The database table shown in Figure 3-6 has two columns designated as foreign keys into some other table or into the same table. When an underscore is used as part of the column name, it indicates that the entry in this column is a foreign key. Most of the time a following underscore indicates a key into another table. When the underscore is in the middle of the column name, this indicates that this key is a reference back into the same table. This syntax does not hold 100% of the time but most of the time it is accurate.

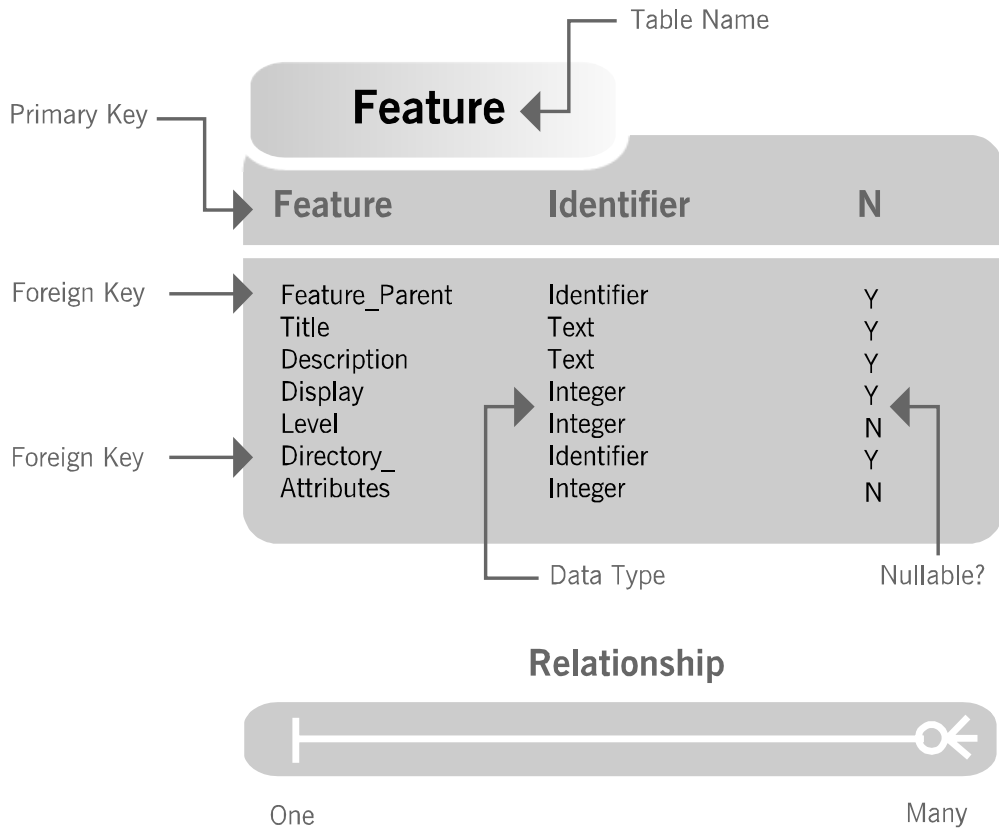


Figure 3-6: Approach used to illustrating the database schema.

Below the diagram of the database table in Figure 3-6 is a representation of how the relationship between tables is shown. Most relationships between tables are a one-to-many relationship. There are some relationships that are one-to-one but there is never a many-to-many relationship permitted in a relational database. A many-to-many relationship is always turned into a one-to-many relationship by the introduction of an additional table.

Now we are ready to look at the database tables that were used to install the Developer Art application.

The Application Design Tables

In these tables the design of the application is defined. The four tables in this category are the Feature, Component, FeatureComponents, and Directory tables. The relationship between these tables is shown in Figure 3-7.

When you design an application, you begin by defining the functionality that the application will provide the person who purchases or uses the product. Next, you group the total functionality of the application into features so that the end user has a choice of whether they want all the functionality or only part of it. The set of features that compose the total functionality of an application is the logical structure of the application. This logical structure of an application can be a hierarchy where features have sub-features and the sub-features have sub-features and so on. This hierarchy is the feature tree for the application. The fact that features can have sub-features is indicated by the one-to-many relationship that the Feature table has with itself.

After defining the logical structure of the application, you then need to create the physical design that will provide the capability as defined by the feature set. The physical design consists of files, registry entries, shortcuts, and other items that comprise what are called components. When the components are created you then have to decide which components are required to implement the functionality of each of the features. Often a component will be needed by more than one feature in order to fulfill its promise of functionality to the end user. In this situation, the components are shared between features of the application.

A situation where many features can share the same component and many components can be used by the same feature is a many-to-many relationship, which is not allowed in a relational database. When a many-to-many relationship appears in the design of a relational database, you need to add an additional table to turn this many-to-many relationship between the two tables to a one-to-many relationship with a third table. This is shown in Figure 3-7 where the relationship between the Feature table and the Component table is implemented through the FeatureComponents table. Figure 3-7 shows that both columns in the FeatureComponents table define the primary key for the table.

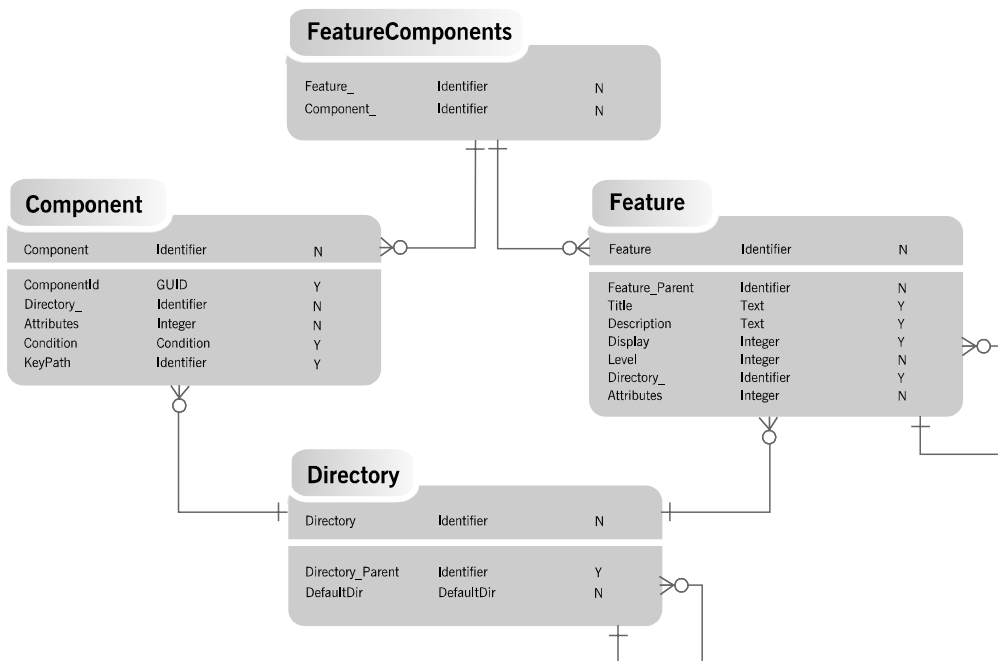


Figure 3-7: The schema of the application design database tables.

The other table that is part of the Application Design Tables category is the Directory table. It may seem a little strange that this table is part of this category and not part of the File Copy category of tables. The reason is that both the Components and the Feature tables have an association with the Directory table. Files are not copied unless the Component with which they are associated is installed. Figure 3-7 shows that the Directory_ column in the Component table is a column that is not allowed to be Null. In other words, all components must have a defined destination on the target machine and that location is defined in the Directory table. The purpose of this reference from the Component table to the Directory table is to enable the copying of files during the installation process.

If you look at the Feature table, you can see that the Directory_ column in this table is allowed to be Null. It is certain that the purpose of this reference from the Feature table to the Directory table is not for copying files. The reason that features can reference an entry in the Directory table is to enable browsing for the location where the feature is to be installed. Typically, in small applications, all features reference the same entry in the Directory table. When the end user browses for an installation

location for a particular feature, they are actually setting the root location for the whole application. Browsing is normally a function provided in the custom setup dialog of an installation's user interface.

Figure 3-7 shows that the Directory table has a one-to-many relationship with itself. This is because directory structures are trees and as such, folders have sub-folders, and so on. The Directory table is one of the more complicated tables in the Windows Installer database. It has to identify both the destination on the target machine and the source location on the distribution media for the purpose of copying files. We will discuss more about how the Directory table works when we get into the how the Windows Installer runs an installation package later in this chapter.

We need to now take a look at what the Project Wizard did with these four tables for the Developer Art application. Remember that throughout the Project Wizard there was no explicit reference to components, only features. When you look at these tables, you can see that the Project Wizard was creating components and giving them default names. It was also creating these components using the rules defined by Microsoft for the proper creation of components. The term that is used by InstallShield Developer for these componentization rules is "Best Practices".

A list of the rules defined by Microsoft for creating components is given below:

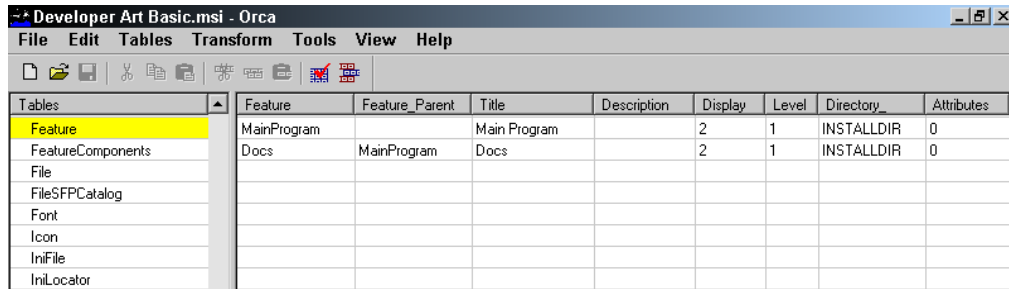
1. Every EXE, DLL, and OCX file needs to be in its own component and this file has to be designated as the key path for the component.
2. Every HLP and CHM file needs to be in its own component. These files need to be designated as the key path for the component. The associated CNT and CHI files need to be added to the component that is installing their associated HLP or CHM file.
3. Never create a component for a file and other resources that is already available in a merge module. Merge modules are discussed in Chapter 14.
4. Every file that serves as the target of a shortcut needs to be in its own component. The target of the shortcut needs to be designated as the key path for the component.
5. Do not install the same resource with two different components. This means files, registry entries, and shortcuts.

The above rules can be considered strong guidelines. If you know what you are doing, you can sometimes break them. It is best to follow these rules unless there are overwhelming reasons to break them. At this time, "Best Practices" include only the first three rules listed. A more detailed discussion of creating components is provided in Chapter 13.

The best way to take a look at these tables as populated by the Project Wizard is to open up the Developer Art Basic.msi file using Orca. From Orca, you can view the database directly and examine the entries in each column of each row. We will only be going to this level of detail with this particular category of tables. This is because this particular set of tables is so important to understanding what you are doing when you create an installation package.

THE FEATURE TABLE

The first table that we will look at is the Feature table. Figure 3-8 shows the rows in the Feature table as displayed in Orca.



Tables	Feature	Feature Parent	Title	Description	Display	Level	Directory	Attributes
Feature	MainProgram		Main Program		2	1	INSTALLDIR	0
FeatureComponents	Docs	MainProgram	Docs		2	1	INSTALLDIR	0
File								
FileSFPCatalog								
Font								
Icon								
IniFile								
IniLocator								

Figure 3-8: *The Feature table as created for the Developer Art application.*

The following list discusses the entries made by the Project Wizard in each of the columns of the Feature table. Since many of these entries deal with how the features are displayed in the custom setup dialog, Figure 3-9 provides a screen capture of this dialog. This will be very handy to refer to as the entries in the Feature table are discussed.

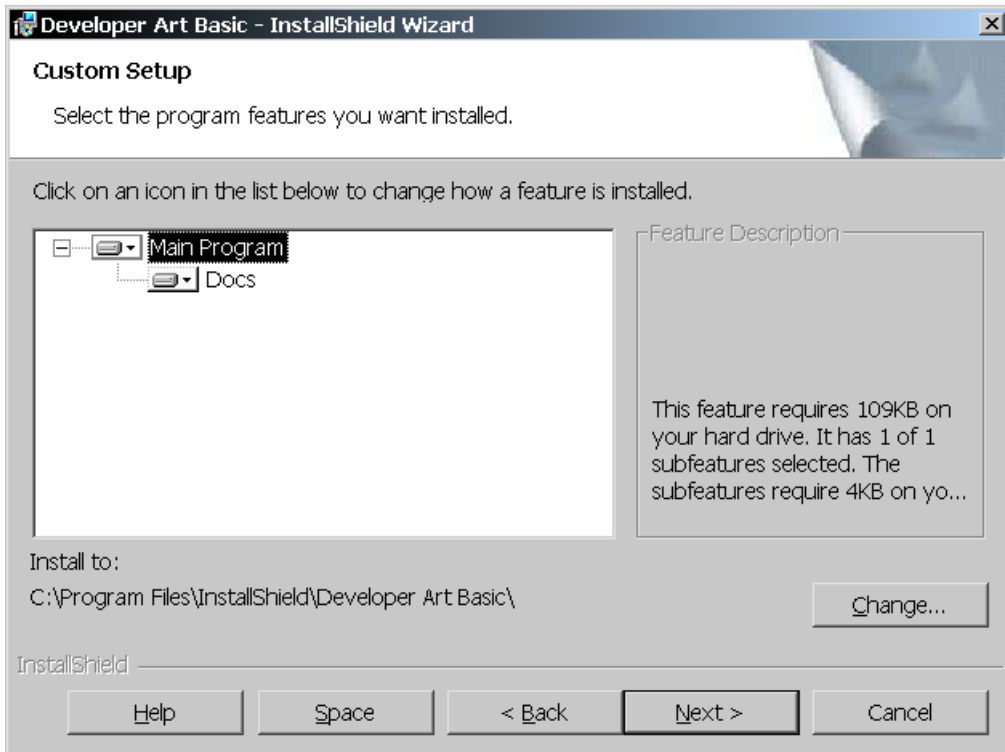


Figure 3-9: *The custom setup dialog in the Developer Art installation user interface.*

Feature: This first column of the Feature table shows the names that were created by the Project Wizard when you entered the display name for each feature in the Application Features dialog. Since the first column is the primary key for the Feature table, these names have to be unique. The Project Wizard would not have created two identical strings. These entries had to conform to the rules for the Identifier data type so this is why these names cannot contain spaces.

Feature_Parent: Figure 3-8 shows one entry in this column. This entry defines that the Docs feature is a sub-feature of the MainProgram feature.

Title: This column contains the names that are displayed in the custom setup dialog. As shown in Figure 3-9, these names are what you entered in the Application Features dialog of the Project Wizard.

Description: This column contains a description for the feature that will be displayed in the custom setup dialog. In Figure 3-9 there is a static text field inside of a group box with the title Feature Description. When an end user highlights a feature, the description entered in this column is displayed in this static text field. You need to use the IDE to enter a description for each feature. What you are seeing in Figure 3-9 is a display of the disk size required by the highlighted feature. When you enter a description for each of these features it will be displayed above the text showing the disk size.

Display: The value in this column determines how the feature tree is first presented when the custom setup dialog is launched. If the value is zero, the feature is hidden and not displayed. If the value is odd, the feature tree is expanded when it is first displayed. This is significant only if there are sub-features under the feature in question. If the value in this column is even, the feature tree is initially collapsed. Figure 3-9 shows this feature tree expanded, but when it was initially displayed it was collapsed because the value in this column is even.

Level: This column defines an *install level* for a feature. Install levels are used to implement simple setup types such as Minimal, Typical, and Complete. Every Windows Installer defines an initial install level by setting the `INSTALLLEVEL` property in the Property table to an integer value from 1 to 32,767. Any feature that has a value in the Level column that is equal to or less than the value defined by the `INSTALLLEVEL` property will be installed by default. A value of 0 for this column disables the feature. The default value of 1 entered in this column is the minimum value that is possible without disabling the feature. The default value of the `INSTALLLEVEL` property is 100.

Directory_: This column is the foreign key into the Directory table that designates the destination for the feature. The entry in this column for both features is `INSTALLDIR`, which is initialized to the default destination folder defined in the Application Information panel of the Project Wizard. In Figure 3-9 below the tree of features, there is a text field with the label “Install to”. This text field displays the present location at which the `INSTALLDIR` property is pointed. You can change this value by clicking on the Change button and browsing to another location. Note that this does not affect the location to where files are copied during installation unless the component destination is also set to the `INSTALLDIR` property.

Attributes: The entry in this column specifies the remote execution options that are available for the feature. This also defines what the default remote execution option will be, in case the end user does not perform a custom setup. To fully understand this entry, you need to be aware of something called feature states. There are four possible states for a feature:

- To be installed to run from the local hard drive.
- To be installed to run from the source media.
- To be advertised so that they are installed on first use.
- To not be installed; that is, the feature will be absent.

The value in this column for both features is 0 and this means that, by default, these features will be installed to run from the local hard drive. Note that the custom setup type dialog in a Standard project is slightly different in appearance and functionality. In particular there is no ability to browse for a specific location to install the feature. This is accomplished using a separate dialog box.

THE COMPONENT TABLE

In the Project Wizard, when you added files to the two features that you created in the Application Features panel, the Project Wizard created components for you. When a file falls into one of the categories defined by the "Best Practices" rules from Microsoft, the component takes on the name of the file that it contains. This file is also made the key path of the component. When a file does not fall under the "Best Practices" rules, such as a text file, the file is placed in a component named AllOtherFilesX where the X represents some sequential number depending on the number of these types of components created. Remember that the component name is the primary key for the Component table and there cannot be duplicate primary keys in any table.

To look in detail at the entries in the Component table for the Developer Art installation database, open the Developer Art Basic.msi file in Orca and select the Component table. The entries for this table are shown in Figure 3-10.

Component	ComponentId	Directory_	Attributes	Condition	KeyPath
AllOtherFiles	{21100822-5B99-4C77-BA9E-12594BE33393}	INSTALLDIR	8		
AllOtherFiles1	{F62320B8-066A-4E47-89FB-A9CA07B06002}	INSTALLDIR	8		
HelpLibrary.dll	{EE457B53-CE26-4886-A95B-A853AD86DAE8}	INSTALLDIR	8		HelpLibrary.dll
DeveloperArt.exe	{6A416DFA-02DE-407F-A264-F17A044800CB}	INSTALLDIR	8		DeveloperArt.exe
ArtWork.dll	{0F941473-C07F-4E23-AF93-1C0FDBC13BA5}	INSTALLDIR	8		ArtWork.dll

Figure 3-10: *The Component table as created for the Developer Art application.*

The following list explains each of the entries made in this table.

Component: This column is the primary key for the Component table. The entries in this column are the Identifier data type and they all have to be different. As described above, the Project Wizard gave the components it created default names that either come from the name of the key path file in the component or follow the AllOtherFilesX format.

ComponentId: This is a very important column because components are registered on the target system through the ComponentId. Here is another instance where a GUID is used. The first instance was in the Summary Information Stream where it was used to make the installation package unique from all other packages. In this case, the GUID is used to make components unique from one another. The GUIDs here for these components were created when you ran the Project Wizard and added files to the features. If you recreated the installation package and added the same files to the same features, a completely different set of GUIDs would appear in the Component table. We will spend a lot of time on this subject in Chapter 13 because the subject of component IDs is so important.

Directory_: This column contains a foreign key into the Directory table just as in the Feature table. Here, however, this reference into the Directory defines the location where the files in the component will be copied as long as the component is installed to the local machine. By default the Project Wizard makes the value in this column the same as the destination folder for the associated feature. This may not always be what you want and, in those cases, you need to make changes in the IDE.

Attributes: This column defines the remote execution options for the component in a very similar fashion as for features. In fact it is this attribute in coordination with the same column in the Feature table that defines what options are available for features in the custom setup dialog. However, this column defines more than just the remote execution options for the component. We will see more of this in Chapter 5.

Condition: This column can contain a conditional statement that controls whether the component is installed. A Null entry in this column is the same as having a condition that evaluates to TRUE. In the Project Wizard there was no opportunity to enter a condition statement so if it is necessary to condition the installation of a component, it has to be done in the IDE.

KeyPath: It is through the value in this column that the Windows Installer knows that the component was installed on the target system or was installed to run from source. For the files that come under the "Best Practices" rules, the entry in this column is the name of the file with the .exe, .dll, .ocx, .hlp, or .chm extension. For components that do not come under the "Best Practices" rules the key path can be the name of a file in the component, it can be a registry entry, a folder, or an ODBC data source name. If this column is left Null, then the key path by default is the folder into which the component is installed. For the Developer Art installation database the three components that were created according to the "Best Practices" rules have the name of the file in this column. For the other components this column is left Null so that the installation folder will be the key path that is registered at the time of installation.

THE FEATURECOMPONENTS TABLE

The FeatureComponents table shows which components in the database are associated with which features (Figure 3-11). For the Developer Art installation this means there are five rows in this table.

Note that both columns form the primary key so this table cannot be built with a Null value in either column. Also note that it is impossible to assign the same component more than once to the same feature because this then would create a duplicate primary key.

Tables	Feature_	Component_
FeatureComponents	MainProgram	AllOtherFiles1
File	MainProgram	HelpLibrary.dll
FileSFPCatalog	MainProgram	DeveloperArt.exe
Font	MainProgram	ArtWork.dll
Icon	Docs	AllOtherFiles
IniFile		

Figure 3-11: The FeatureComponents table as created for the Developer Art application.

THE DIRECTORY TABLE

The fourth and final table in the Application Design category is the Directory table (Figure 3-12).

Tables	Directory	Directory_Parent	DefaultDir
Directory	TARGETDIR		SourceDir
DiLocator	ISCommonFilesFolder	CommonFilesFolder	Instal~1\InstallShield
DuplicateFile	INSTALLDIR	DEVELOPER_ART_BASIC	.
Environment	DEVELOPER_ART_BASIC	INSTALLSHIELD	DEVELO~1\Developer Art Basic
Error	ISUpdateServiceFolder	ISCommonFilesFolder	UPDATE~1\UpdateService
EventMapping	ISYourProductDir	ISYourCompanyDir	YOURPR~1\Your Product Name
Extension	INSTALLSHIELD	ProgramFilesFolder	INSTAL~1\InstallShield
Feature	ISYourCompanyDir	ProgramFilesFolder	YOURCO~1\Your Company Name
FeatureComponents	AdminToolsFolder	TARGETDIR	..\Admin~1\AdminTools
File	AppDataFolder	TARGETDIR	..\APPLIC~1\Application Data
FileSFPCatalog	CommonAppDataFolder	TARGETDIR	..\Common~1\CommonAppData
Font	CommonFiles64Folder	TARGETDIR	..\Common64
Icon	CommonFilesFolder	TARGETDIR	..\Common
IniFile	DesktopFolder	TARGETDIR	..\Desktop
IniLocator	FavoritesFolder	TARGETDIR	..\FAVORI~1\Favorites
InstallExecutesSequence	FontsFolder	TARGETDIR	..\Fonts
InstallUISequence	GlobalAssemblyCache	TARGETDIR	..\Global~1\GlobalAssemblyCache
IsolatedComponent	LocalAppDataFolder	TARGETDIR	..\LocalA~1\LocalAppData
LaunchCondition	MyPicturesFolder	TARGETDIR	..\MyPict~1\MyPictures
ListBox	PersonalFolder	TARGETDIR	..\Personal
ListView	PrimaryVolumePath	TARGETDIR	..\Primar~1\PrimaryVolumePath
LockPermissions	ProgramFiles64Folder	TARGETDIR	..\Prog64~1\Program Files 64
MIME	ProgramFilesFolder	TARGETDIR	..\PROGRA~1\program files
Media	ProgramMenuFolder	TARGETDIR	..\Programs
MoveFile	SendToFolder	TARGETDIR	..\SendTo
MsiAssembly	StartMenuFolder	TARGETDIR	..\STARTM~1\Start Menu
MsiAssemblyName	StartupFolder	TARGETDIR	..\Startup
MsiDigitalCertificate	System16Folder	TARGETDIR	..\System
MsiDigitalSignature	System64Folder	TARGETDIR	..\System64

Tables: 84 Directory - 34 rows No column is selected.

Figure 3-12: The Directory table as created for the Developer Art application.

This table is one of the more complex tables in the database. We will get into all the details of this table when we discuss the operation of the Windows Installer later in this chapter.

The first two columns of this table are identifiers and the `Directory_Parent` column is a key back into the `Directory` column. During an installation, the Windows Installer assesses the amount of disk space required by the application being installed. During this operation the `Directory` table is resolved. What this means is that every entry in the `Directory` column of the `Directory` table is resolved to have an absolute location on the target machine. Also, every entry in the first column has an absolute location on the distribution media. In other words every identifier in the `Directory` column of the `Directory` table provides both the source and destination locations required to perform the file copy operation. How this resolution operation is implemented is discussed in the section entitled *How Does The Windows Installer Perform an Installation?*

The File Copy Tables

There are only three tables in this group and one of them we have already covered as part of the Application Design set of tables. The tables in this category are the `File`, `Component`, and the `Media` tables. The duplicate table from the previous category is the `Component` table. The relationship between these tables is shown in Figure 3-13.

The `File` table lists all the files in an application and it has a one-to-many relationship with the `Component` table, as shown in Figure 3-13. This relationship indicates that there can be many files in a component, but more than one component cannot install the same file as defined by the identifier in the first column of the `File` table. In other words files can be shared among components. This is in accord with rule number five for creating components, which says that the same resource cannot be installed with two different components.

What is interesting about this is that, in the `File` table, the name of the file is not the primary key. It is an identifier instead. The name of the file is provided in the `FileName` column and this column is not part of the primary key definition. It is possible to create two different identifiers for the same file name and thus share the same file name among different components. This is only applicable if two files have the same name but they are actually different files. As an example, an installation might have two `Readme.txt` files with one of them in English and one of them in German.

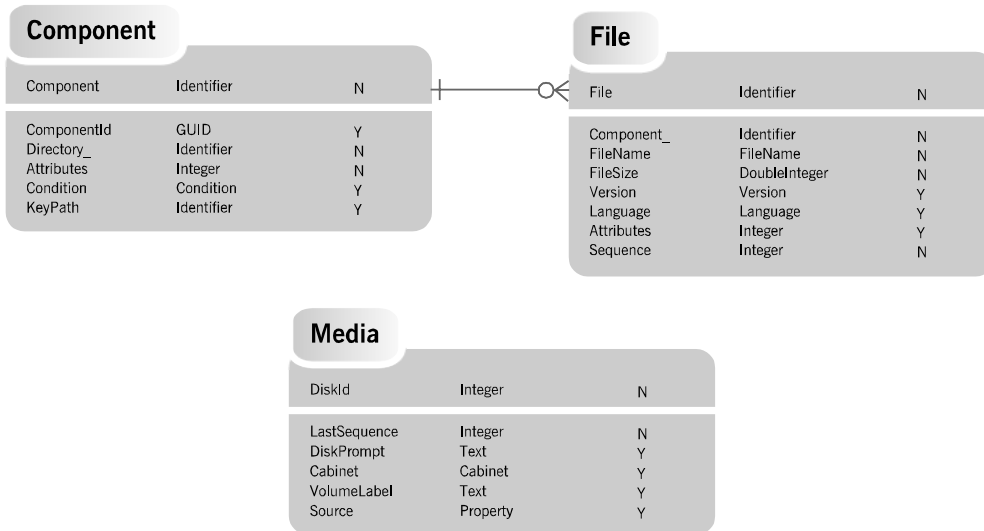


Figure 3-13: *The schema of the file copy database tables.*

In Figure 3-13, the Directory table does not appear as a member of this category of tables. Without the Directory table how does the installation know where to copy the files listed in the File table? The answer is that the File table connects to the Directory table through the Component table. A file listed in the File table will be copied to the destination of the associated component if that component is going to be installed. A component is installed if the associated feature is to be installed and the condition on the component evaluates to TRUE.

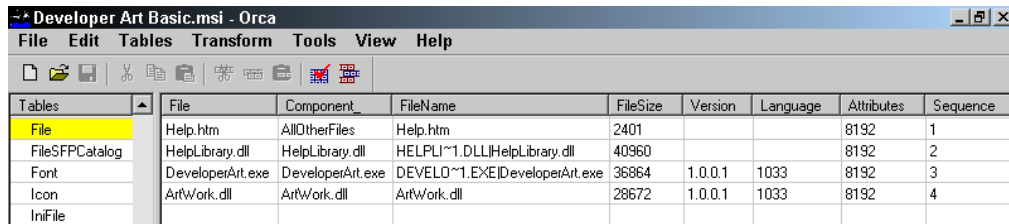
Figure 3-13 shows that the Media table has no direct relationship with either the File table or the Component table. The purpose of the Media table is to describe the disk set that makes up the distribution media for the application. Each disk in the distribution media has a row in the Media table and the rows have to be in the order of the disk numbers. When an application is composed of more than one disk, the file copy mechanism needs to know on which disk each of the files is located. This is accomplished through the entry in the LastSequence column. In the Media table's first row, the value in the LastSequence column is the value from the Sequence column of the File table for the last file that is being shipped on the first disk. In the second row of the Media table, the value in the LastSequence column is the value from the Sequence column of the File table for the last file on the second disk. The files on the second disk then will include all the files listed in the File table that have

sequence numbers greater than the LastSequence value in the first row of the Media table and less than or equal to the LastSequence value in the second row of the Media table.

Even though there is no database relationship between the Media table and the File table, there is a functional relationship. When building the distribution image for a multi-disk application, the values entered into the File table have to be synchronized with the values that are entered into the Media table.

THE FILE TABLE

The rows created in the File table for the Developer Art application are shown in Figure 3-14.



Tables	File	Component	FileName	FileSize	Version	Language	Attributes	Sequence
File	Help.htm	AllOtherFiles	Help.htm	2401			8192	1
FileSFPCatalog	HelpLibrary.dll	HelpLibrary.dll	HELPLI~1.DLL HelpLibrary.dll	40960			8192	2
Font	DeveloperArt.exe	DeveloperArt.exe	DEVELO~1.EXE DeveloperArt.exe	36864	1.0.0.1	1033	8192	3
Icon	ArtWork.dll	ArtWork.dll	ArtWork.dll	28672	1.0.0.1	1033	8192	4
IniFile								

Figure 3-14: *The File table as created for the Developer Art application.*

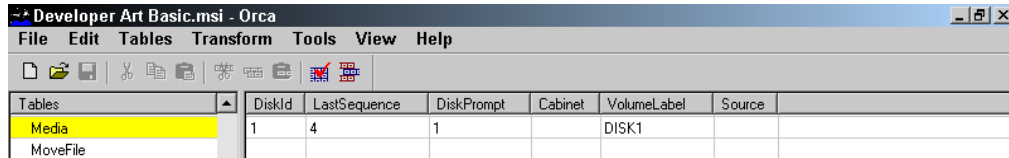
Figure 3-14 shows that the Project Wizard uses the name of the file as the primary key. If a file name has spaces in it, the space is replaced with the underscore (_) character before it is used as the primary key. The second column identifies the component that will install the file. The values in the Version and Language columns are extracted from the version resource of the file. When a file does not have a version resource, these columns are left Null unless you launch the project's Properties dialog and add your own version and language that will then be built into the File table. You can access the Properties dialog from the Application Files panel in the Project Wizard by right clicking on a file and selecting the Properties option. This operation was covered in Chapter 2.

The Attributes column is where the attributes for the file are specified. For example, if a file is to be Read-Only after it is installed, this is where that attribute is specified. The Project Wizard assigned an attribute of 8192 for all the files in the Developer Art installation package. This attribute value means that the files are in an uncompressed format on the distribution media. The final column of the File table shows the

sequence in which the files will be copied to the target system. It also identifies on which disk of the distribution media the file resides, as discussed above.

THE MEDIA TABLE

Because the Developer Art application is so small it requires only one disk to hold it, there is only one row in the Media table (Figure 3-15).



Tables	DiskId	LastSequence	DiskPrompt	Cabinet	VolumeLabel	Source
Media	1	4	1		DISK1	
MoveFile						

Figure 3-15: *The Media table as created for the Developer Art application.*

The entry in the DiskId column for this one and only row is 1. Since there are only four files that make up this application and the largest value in the Sequence column of the File table is 4, the value in the LastSequence column of the Media table is also set as 4. In the DiskPrompt column, the Project Wizard has inserted the number 1. This column has a Text data type and, as such, this column can be modified to be a string that matches the text printed on the disk. This text string is used to prompt the user when this particular disk is required. The Cabinet column is Null because the files are not compressed into a cabinet file. The VolumeLabel column contains the text string DISK1. However, this column has no meaning if there is only one disk in the distribution media. This column is used to verify that when a user has been prompted to put in the next disk that the correct disk has been inserted.

The Registry Entry Tables

There are seven tables in this category. They are the Registry, Class, Progid, TypeLib, Feature, Component, and Directory tables. These are not the only tables that make entries in the registry but they are the only ones that come into play for the Developer Art installation package. Figure 3-16 shows the schema diagram for this category of tables.

All the registry entries that are required for the Developer Art application relate to the fact that the ArtWork.dll file is a COM server. This topic is covered in depth in Chapter 14, so it is not necessary to look at the COM-related registry entries required

for this particular file. Note, however, that once again some of the tables from the Application Design category appear in this category. The Application Design tables are the core tables of an installation database.

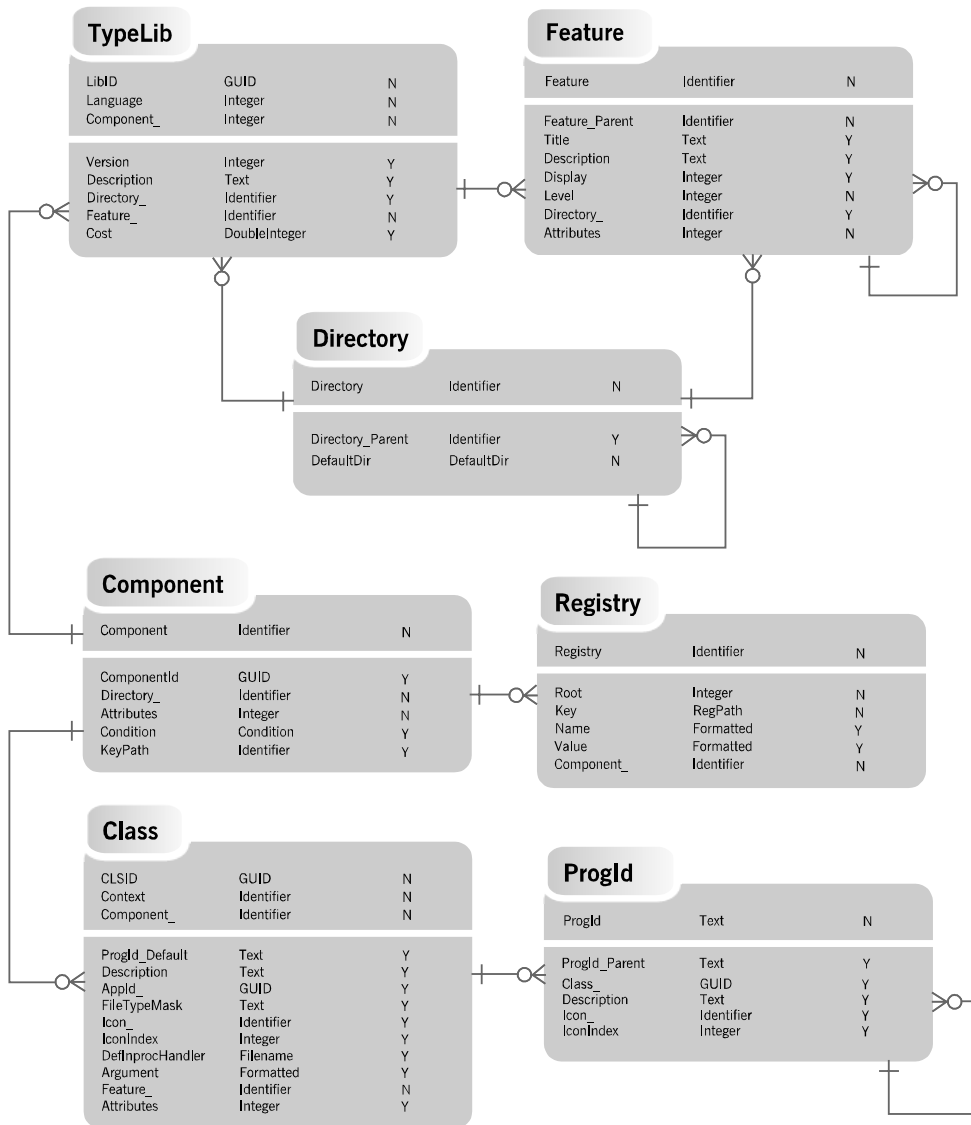


Figure 3-16: The schema of the registry entry database tables.

The Installation Procedure Tables

There are eight tables in this group and all of these tables are used to control the installation process. Six of these tables are called the sequence tables. The other two tables in this category are the Property and CustomAction tables. The schema diagram for all of these tables is shown in Figure 3-17.

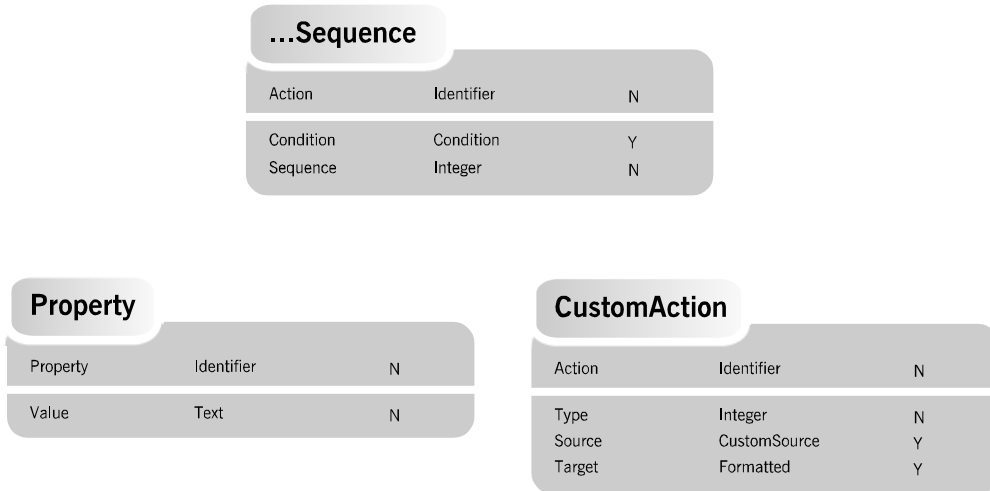


Figure 3-17: *The schema of the installation procedure database tables.*

Understanding these tables requires a brief introduction to the how the Windows Installer implements an installation after the MSI package has been passed to it.

The Windows Installer knows how to implement three types of installations. These three types of installations are called top-level actions and one of these top-level actions is implemented depending on what command line arguments are passed to the Windows Installer engine. There are two sequence tables that come into play for each of the three top-level actions thus the need for a total of six sequence tables in the database. As shown in Figure 3-17, each sequence table has three columns with the names Action, Condition, and Sequence. The following list provides the names of the three top-level actions and the names of the two sequence tables that come into play for that top-level action.

INSTALL: This top-level action is called to install or remove components. This action queries the `InstallUISequence` table and the `InstallExecuteSequence` table for the actions to execute, the condition for action execution, and the place of the action in the sequence of actions.

ADMIN: This top-level action is called to perform an administrative installation. This action queries the `AdminUISequence` table and the `AdminExecuteSequence` table for the actions to execute, the condition for action execution, and the place of the action in the sequence of actions.

ADVERTISE: This top-level action is called to install or remove advertised components. This action queries the `AdvtUISequence` table and the `AdvtExecuteSequence` table for the actions to execute, the condition for action execution, and the place of the action in the sequence of actions. For this top-level action, the Windows Installer does not use the `AdvtUISequence` table. It is in the database schema for purposes of symmetry only. Advertising refers to the installer's ability to provide the launching interfaces of an application without physically installing the application. The installer does not install the necessary components until a user or application activates an advertised interface. This concept is called install-on-demand.

The UI sequence tables allow three types of actions. These are standard actions, custom actions, and dialogs. In the execute sequence tables, only standard actions and custom actions are permitted. The Windows Installer executes the actions listed in the `Action` column of a sequence table in the order of the sequence number in the `Sequence` column. The action is executed if the condition in the `Condition` column evaluates to `TRUE`. A Null entry in the `Condition` column is considered a `TRUE` condition.

Standard actions can be considered the built-in functions provided by the Windows Installer. Custom actions provide a way for setup developers to extend the functionality of the Windows Installer. Dialogs are the user interface as defined in the various tables of the database. Figure 3-18 shows the `InstallUISequence` table in the Developer Art installation database. The Project Wizard creates the entries in the sequence tables based on a default recommended by Microsoft.

Tables	Action	Condition	Sequence
InstallUISequence	SetupCompleteError		-3
IsolatedComponent	SetupInterrupted		-2
LaunchCondition	SetupCompleteSuccess		-1
ListBox	AppSearch		25
ListView	LaunchConditions		50
LockPermissions	SetupInitialization		100
MIME	FindRelatedProducts		150
Media	CCPSearch	CCP_TEST	250
MoveFile	RMCCPSearch	Not CCP_SUCCESS And CCP_TEST	300
MsiAssembly	ValidateProductID		350
MsiAssemblyName	CostInitialize		400
MsiDigitalCertificate	FileCost		450
MsiDigitalSignature	IsolateComponents		500
MsiFileHash	ResolveSource	Not Installed	525
ODBCAttribute	CostFinalize		550
ODBCDataSource	ISInitAllUsers	VersionNT And NOT Installed	575
ODBCDriver	MigrateFeatureStates		600
ODBCSourceAttribute	PatchWelcome	PATCH	625
ODBCTranslator	InstallWelcome	Not Installed And Not PATCH	662
Patch	SetupResume	Installed And (RESUME Or Preselected) And Not PATCH	700
PatchPackage	MaintenanceWelcome	Installed And Not RESUME And Not Preselected And Not PATCH	750
ProgId	SetupProgress		800
Property	ExecuteAction		850
PublishComponent			
RadioButton			
RegLocator			
Registry			
RemoveFile			
RemoveIniFile			

Tables: 84 InstallUISequence - 23 rows No column is selected.

Figure 3-18: *The InstallUISequence table as created for the Developer Art application.*

The Property table contains the global variables of the installation package. There are two major types of properties, private and public. A screen capture of part of the Property table for the Developer Art installation database is shown in Figure 3-19.

The Property column of this table contains some names that are mixed-case letters and other names that are in all upper-case letters. The property names that are in mixed-case letters are called private properties and the properties that have all upper-case letters are called public properties. One of the major uses of public properties is that they can have their values set at the command line whereas private properties cannot. We will discuss more about properties and their uses throughout the book.

The screenshot shows the Orca application window titled "Developer Art Basic.msi - Orca". The "Property" table is selected and highlighted in yellow. The table contains various properties and their values, including application-specific details and system-related settings. The status bar at the bottom indicates "Tables: 84", "Property - 40 rows", and "No column is selected."

Table	Property	Value
ODBCDataSource	ARPCOMMENTS	Your Comments
ODBCDriver	ARPCONTACT	Customer Support Department
ODBCSourceAttribute	ARPHHELP	http://www.yourcompany.com/help
ODBCTranslator	ARPHHELPTELEPHONE	911
Patch	ARPURLINFOABOUT	http://www.installshield.com
PatchPackage	ARPURLUPDATEINFO	http://www.yourcompany.com/updateinfo
ProgId	AgreeToLicense	No
Property	ApplicationUsers	AllUsers
PublishComponent	DWUSINTERVAL	30
RadioButton	DefaultUIFont	Tahoma8
RegLocator	DialogCaption	InstallShield for Windows Installer
Registry	DiskPrompt	[1]
RemoveFile	DisplayNameCustom	Custom
RemoveIniFile	DisplayNameMinimal	Minimal
RemoveRegistry	DisplayNameTypical	Typical
ReserveCost	Display_IsBitmapDlg	1
SFPCatalog	ErrorDialog	SetupError
SelfReg	ISINSTALLLEVEL	100
ServiceControl	ISCHECKFORPRODUCTUPDATES	1
ServiceInstall	ISSCRIPT_VERSION_MISSING	The InstallScript engine is missing from this machine. If available, please run ISScript.r
Shortcut	ISSCRIPT_VERSION_OLD	The InstallScript engine on this machine is older than the version required to run this se
Signature	InstallChoice	AR
TextStyle	Manufacturer	InstallShield Software Corporation
TypeLib	PIDTemplate	12345<###-%-%-%-%-%>@%@@@
UIText	ProductCode	{8583687F-CC74-4CC6-AF1D-EA3365B6914D}
Upgrade	ProductID	none
Verb	ProductLanguage	1033
_Validation	ProductName	Developer Art Basic

Figure 3-19: *The Property table as created for the Developer Art application.*

Even though Figure 3-19 shows many properties in the Property table, there are only five properties that are required for each installation database. These required properties are the ProductCode, ProductLanguage, Manufacturer, ProductVersion, and ProductName properties. The ProductCode property is a GUID that is created by the Project Wizard. The ProductLanguage property is defaulted to English, and the other three properties are set by the entries that were made in the Application Information and Company Information panels in the Project Wizard.

Finally, the CustomAction table is where all user-defined actions are listed. For the Developer Art application the Project Wizard inserts three custom actions as shown in Figure 3-20. The Project Wizard inserts the ISInitAllUsers custom action in order to make the default a per-machine installation. This is done because on the General tab in the Options dialog the "Automatically create ISSetAllUsers action" is selected by default and you did not deselect this before running the Project Wizard in Chapter 2. The other two custom actions are placed in the CustomAction table in case you

want to enable the InstallShield Update service. When you ran the Project Wizard in Chapter 2 you deselected this option so these custom actions are not inserted into the any of the sequence tables.

The screenshot shows the Orca application window titled "Developer Art Basic.msi - Orca". The menu bar includes File, Edit, Tables, Transform, Tools, View, and Help. Below the menu bar is a toolbar with various icons. The main area displays a table with the following data:

Tables	Action	Type	Source	Target
CustomAction	ISInitAllUsers	307	ALLUSERS	2
Dialog	CheckForProductUpdates	226	ISUpdateServiceFolder	[[ISUpdateServiceFolder]agent.exe "/au[ProductCo
Directory	CheckForProductUpdatesOnReboot	226	ISUpdateServiceFolder	[[ISUpdateServiceFolder]agent.exe "/au[ProductCo
DirLocator				

Figure 3-20: *The CustomAction table as created for the Developer Art application.*

The User Interface Tables

Since Chapter 12 is devoted to the creation of the user interface for installation packages, this section provides just a brief discussion of the schema diagram for the tables that came into play for the Developer Art installation package. There are other tables in the database that relate to the creation of a user interface and but they are not used for the Developer Art installation user interface.

Ten tables were required to define the user interface for the Developer Art installation. Figure 3-21 shows the schema diagram for these tables. One of the first impressions that you might have from looking at the diagram in Figure 3-21 is that the creation of a user interface inside the database is very complicated. It is complicated, but the Dialog Editor in InstallShield Developer makes it easier by doing most of the work for you. However, you need a solid understanding of the user interface tables in order to create new dialogs or modify existing dialogs.

This section introduces the user interface tables shown in Figure 3-21. Most of these tables are already populated with values prior to running the Project Wizard. These tables are built in to the templates that are used to create projects generated by the Project Wizard, as well as new projects created directly in the IDE.

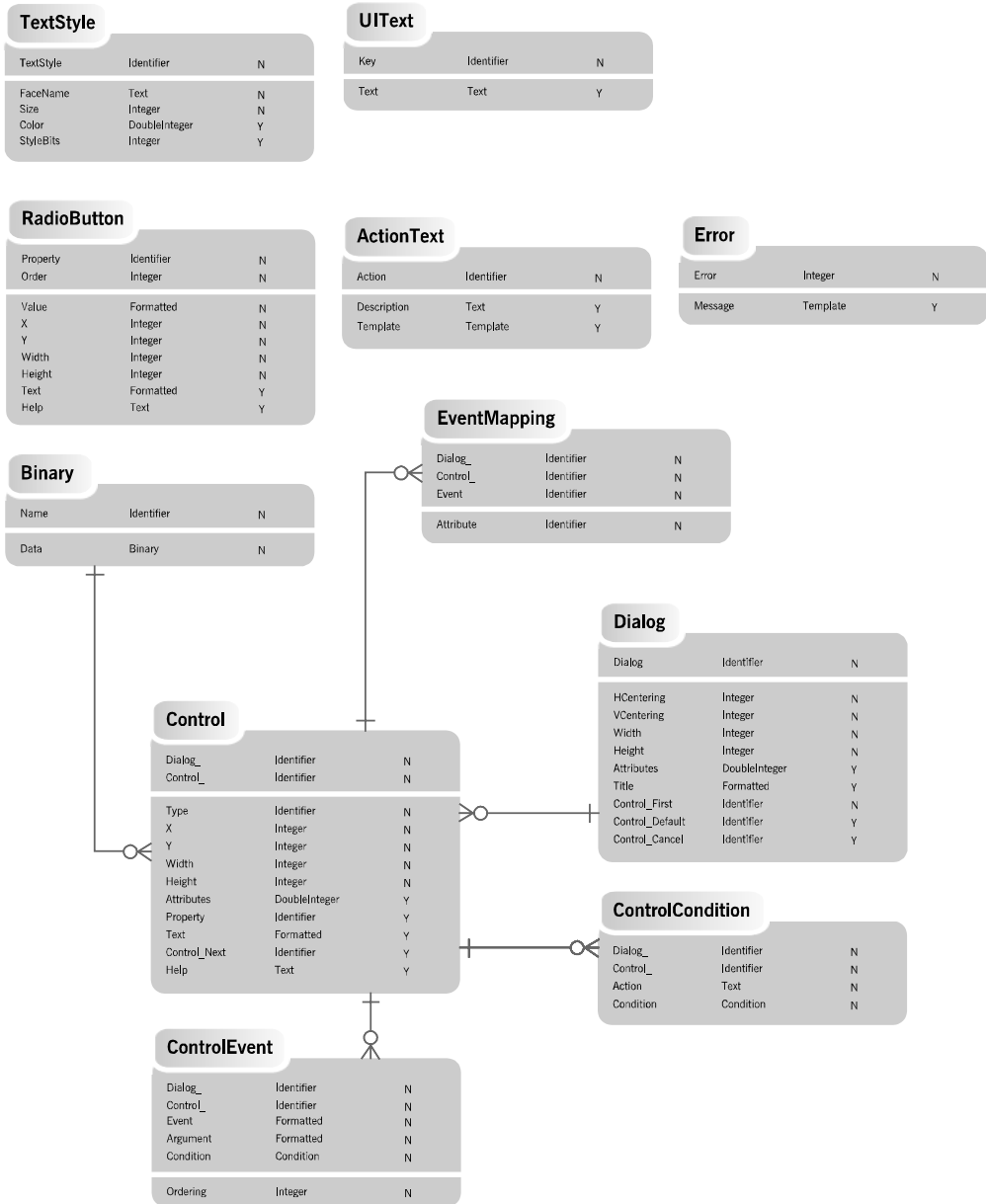


Figure 3-21: The schema of the Developer Art user interface database tables.

The following list describes each of the tables shown in Figure 3-21.

ActionText Table: The ActionText table contains text that is displayed in the progress dialog box, which informs the end user of the installation’s progress. The text strings that are displayed in the progress dialog can also be written to a log file if one is created during the installation. The first column of this table is the name of an action. When the Windows Installer executes the action listed in this column, the description in the second column is displayed. For this description to be displayed it is necessary to author some rows in the EventMapping table.

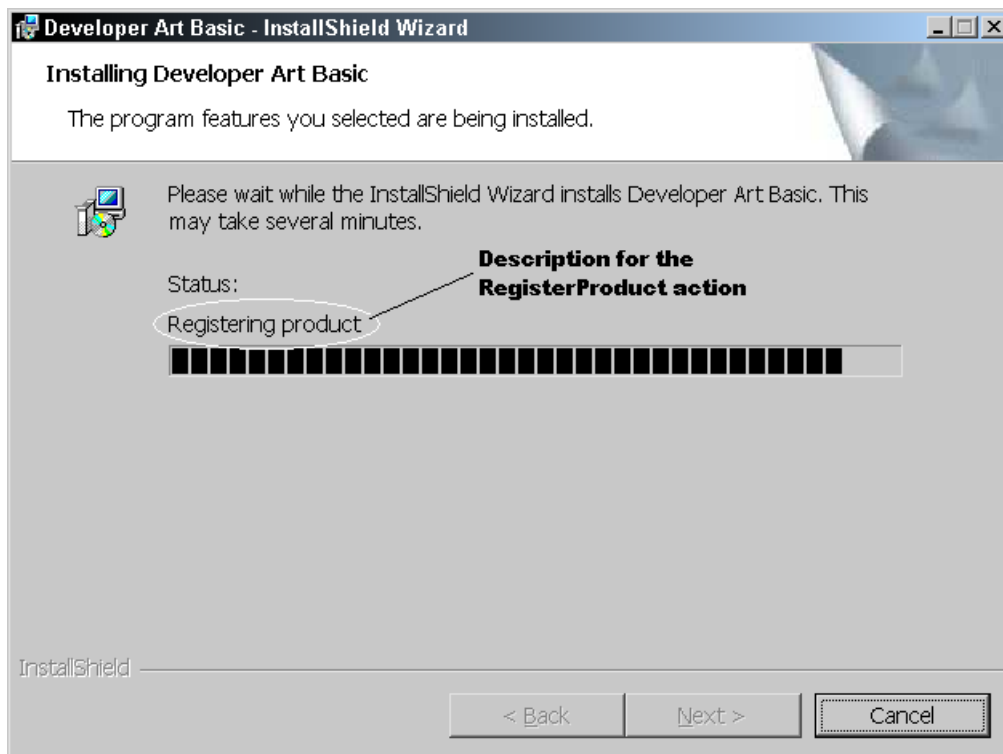


Figure 3-22: *The progress dialog for the Developer Art installation.*

This will be covered in greater detail in Chapter 12 on creating the user interface. The third column of this table, if used, can display data messages about the action that is running. For example data messages can be file names, file sizes, and directory names. The default user interface that is created by InstallShield

Developer does not enable the display of data messages for any actions, but it does show the description associated with each action in this table (Figure 3-22).

Binary Table: The Binary table holds all the graphics used in the user interface. This table can hold many other items but, by default, the graphics used in the dialogs provided by InstallShield Developer are always in this table. They are included in this table as binary streams. When the Windows Installer displays a dialog that contains a graphic image, it streams this image directly out of the Binary table into the dialog control that is used to hold the image. No intermediate file is created.

Control Table: The Control table defines the controls that appear on each dialog in the user interface. The Windows Installer implements a number of standard controls that setup developers can place on a dialog. However, not all standard Microsoft Windows controls are available and custom controls cannot be created for use with the Windows Installer user interface. A control is created from a template recorded in the control table. This template is slightly different than templates used in normal Windows user interface programming in that it contains the unique name of the dialog box on which the control appears.

ControlCondition Table: The ControlCondition table enables you to specify special actions to be applied to controls based on the result of a condition statement. For example, the Next button can be conditioned so that under one set of circumstances it displays one dialog, but under another set of circumstances it displays a different dialog.

ControlEvent Table: The ControlEvent table allows you to specify what happens when the end user interacts with any control within a dialog box. For example, a click of a push button can be defined to trigger a transition to another dialog box, to exit a dialog box sequence, or to begin the installation process.

Dialog Table: The Dialog table contains all the dialogs that appear in the user interface. The process of creating a dialog box in Windows Installer is similar to the process of creating a dialog box programmatically using a Microsoft Windows API dialog box template. The Windows Installer stores the dialog box parameters in the Dialog table. The Dialog table contains an attributes column that is analogous to Window styles in the Microsoft Windows user interface API. However, the number of Dialog Style Bits in Windows Installer is a reduced and specialized set.

Error Table: The Error table is used to look up error message formatting templates when processing errors with an error code set. When a standard action runs into a problem during an installation, it sends an error number back to the Windows Installer. The Windows Installer locates the error in the Error table and displays the error message that is entered in the second column.

EventMapping Table: The EventMapping table lists the controls that subscribe to some control event and lists the attribute to be changed when the event is published. The example that was discussed under the topic of the ActionText table is what this table is used for. The main use of this table is to display information in the progress dialog and in the custom setup dialog when the end user highlights a feature name and the description, size, and destination are displayed in static text controls in the dialog.

RadioButton Table: This table is used to define the radio button controls in all radio button groups in the user interface. Note that the first column in this table is the name of a property. It is this property that ties a particular set of radio buttons together into a group. There are other similar tables for list box, combo box, and list view controls. These types of controls are not part of the standard set of dialogs that are available in a project. If these types of controls are required, they have to be added using the InstallShield Developer Dialog Editor.

TextStyle Table: The TextStyle table defines the different font styles used in controls that display text. You can define more styles as required. Text can be defined in different colors, as well as different styles such as bold, italicized, and underlined.

UIText Table: The UIText table contains the localized versions of some of the strings used in the user interface. These strings are not part of any other table. The UIText table is for strings that have no logical place in any other table. A good example of where these strings are used is in the custom setup dialog where the permissible install states for a feature can be selected by clicking on the down arrow on the feature icon.

This has just been a brief introduction into what goes into creating a user interface and defining it inside the database tables. The next section discusses the tables that implement the integration of the application with the target machine's desktop.

The Desktop Integration Tables

Desktop integration is the operation that creates the means for the end user to easily launch the application that was installed. This has to do with the creation of shortcuts to the application's executable or executables. The most common locations to place these shortcuts are on the Start\Programs menu and the desktop. If an application has more than one shortcut these are commonly grouped together inside a folder that is placed on the Start\Programs menu. There are four tables that are involved in the creation of shortcuts during an installation (Figure 3-23).

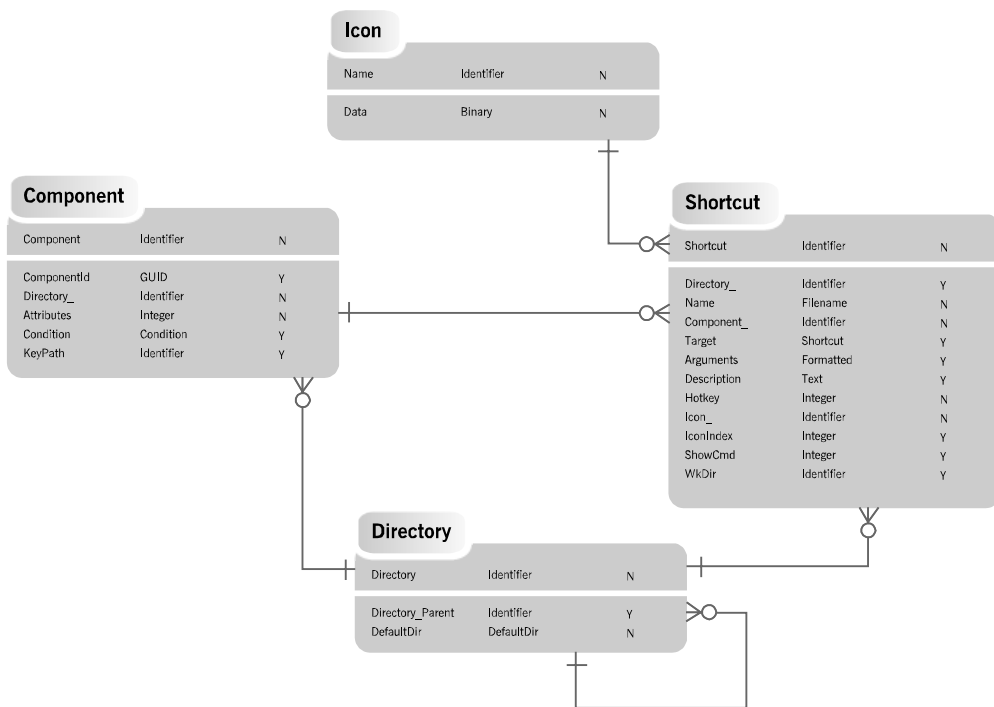


Figure 3-23: The schema of the Developer Art desktop integration database tables.

There are only two new tables in this group of tables from what we have seen in the other table groups. The Windows Installer has added a new type of shortcut that provides a new way to integrate an application with the desktop. The standard shortcut that has always been around is still available. The standard shortcut can be edited to use a specific icon and to point to a particular target. The new type of shortcut, called an MSI shortcut or an advertisable shortcut, is different in that it

contains additional information about the application to which it points. This new information contained in the shortcut is called a Darwin Descriptor List and it has information about the product, features, and components that compose the product. This new type of shortcut cannot be edited.

The tables that implement the creation of shortcuts during an installation are discussed in the following list:

Shortcut Table: The Shortcut table contains the information the installation requires to create the shortcuts on the target machine. Figure 3-23 shows that there are foreign keys into the Directory, Component, and the Icon tables. The link to the Directory table specifies in which folder the shortcut is to be created. The link to the Component table indicates that the shortcut will be created only if the component is being installed or removed if the component is being uninstalled. The link to the Icon table indicates the binary stream that is used to provide the icon for the shortcut.

The Target column of this table determines whether the Windows Installer creates an MSI shortcut or a standard shortcut. The entry in this column can be either a text string that evaluates to the absolute path to the shortcut's target or it can be a foreign key into the Feature table. When the entry in the Target column is a foreign key into the Feature table, an MSI shortcut is created and the target of this shortcut is the file identified as the key path of the component in the Component_ column. Figure 3-24 shows on the left the Properties dialog for the standard shortcut created by the Developer Art installation and on the right what the shortcut properties would be if you had created an MSI shortcut. Remember from Chapter 2 that the Project Wizard only creates standard shortcuts.

Looking at Figure 3-24 and comparing the two Properties dialogs that are shown we can see that there are two major differences between a standard shortcut shown on the left and an MSI shortcut shown on the right. The first of these differences is that we can modify the target at which a standard shortcut points but we cannot do this for an MSI shortcut. The second major difference is that for a standard shortcut we can change the icon that is used but not for an MSI shortcut. Also, if we were to compare the sizes of the two types of shortcuts we would see that the MSI shortcut is larger. This is because only an MSI shortcut can be advertised and the larger size is a result of the additional information that it contains in order to support advertisement. Advertisement, however, requires that a specific version of the shell be installed on the target machine. The versions

of the shell that come with Windows XP, Windows 2000, and Windows 98 support MSI shortcuts. Earlier operating systems need to have, at a minimum, Internet Explorer 4.01 with service pack 1 installed in order to support MSI shortcuts. If the shell does not support MSI shortcuts, the Windows Installer creates a standard shortcut regardless of how it has been defined in the Shortcut table.

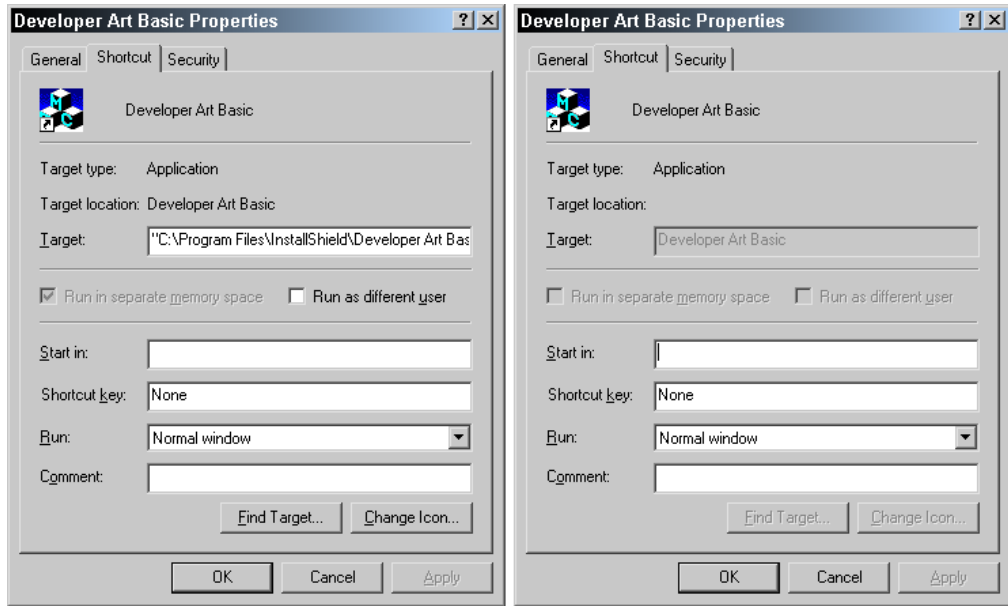


Figure 3-24: *The Developer Art shortcut Properties dialogs in Windows Explorer for both a standard shortcut (left) and an MSI shortcut (right).*

Icon Table: The Icon table is like the Binary table in that it holds files as binary streams. In this case the binary streams consist of files that are icon files or resource files that contain icon resources. In Windows Installer the icons used for shortcuts need to be in separate files because of advertisement. When the icon is used for a shortcut the icon has to be in a file that has the Portable Executable (PE) format such as an executable or a dynamic link library. There is one more rule that needs to be followed for this resource file and that is the extension of the resource file has to be the same as the extension of the file that is the target of the shortcut. Remember that when an application is advertised the application is registered in the registry but no files are copied to the target system. However, for the advertised application to display a shortcut with an associated icon on the

Start\Programs menu there needs to be an icon source copied to the system. This icon source comes from the Icon table.

The Installation Validation Table

The Validation table is the only temporary table that is left in the database after a build of the installation package is completed. The purpose of this table is to enable the running of an internal validation and an internal consistency evaluation (ICE). This table contains the column names and the allowable column values for all the tables in the database. Figure 3-25 shows the schema diagram for this table.

_Validation		
Table	Identifier	N
Column	Identifier	N
Nullable	Text	N
MinValue	DoubleInteger	Y
MaxValue	DoubleInteger	Y
KeyTable	Identifier	Y
KeyColumn	Integer	Y
Category	Text	Y
Set	Text	Y
Description	Text	Y

Figure 3-25: *The schema of the _Validation database table.*

When authoring an installation database programmatically, this table makes it impossible to build the database incorrectly relative to the type and values of data placed into each column. This validation happens at build time and, if things are not correct, a build error is generated. The internal consistency evaluation (ICE) occurs after the build is complete. You have to initiate the internal consistency validation by using one of the various validation suites created by Microsoft. You can run the internal consistency evaluation from the InstallShield Developer IDE by accessing the Validation option from the Build pull-down menu (Figure 3-26).

Running a successful validation is required for any application to receive the "Certified for Windows" logo. If you want this logo, but do not determine whether your application can be validated without error before submission of your application for testing, then you could lose money if the application gets to VeriTest and it fails to pass validation. This is the first test that is run on the application and, if it fails, no further testing is done. Even if you are not interested in obtaining the logo, it is highly recommended that running one of the validation suites become part of your build process. Validation can catch many items that cause problems for the end user.

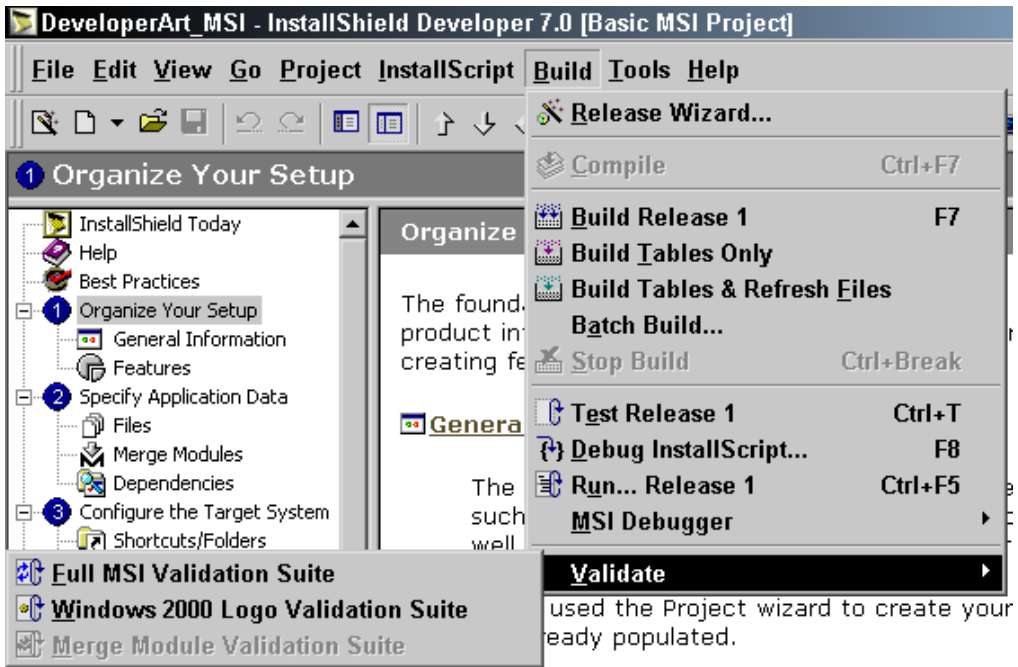


Figure 3-26: *The Validation option on the Build pull-down menu.*

Internal consistency evaluation is implemented by the creation of *internal consistency evaluators* that are stored in an ICE database. An ICE database is stored in an MSI database that has a .cub extension. An ICE is a custom action that interrogates the database in order to find incorrect relationships between tables. Microsoft has provided many ICEs but it is also possible to create a custom CUB (pronounced cube) database. There are three Microsoft-created CUB files: one for logo purposes, one that is more complete and contains tests for newer tables, and one for merge modules. To understand more about these validation routines, see the Windows

Installer help file. The topic of creating internal consistency evaluators is outside the scope of this book.

How Does the Windows Installer Perform an Installation?

As discussed, the Windows Installer knows how to perform three different types of installations as defined by the three top-level actions. This section examines the `INSTALL` top-level action to see what happens during an installation or uninstallation. Everything discussed here that relates to the process of the `INSTALL` top-level action can be applied to the understanding of the `ADMIN` and the `ADVERTISE` top-level actions.

When Windows Installer performs the `INSTALL` top-level action, the Windows Installer first queries the `InstallUISequence` and the `InstallExecuteSequence` tables to determine what actions need to be executed. For the `ADMIN` top-level action, the Windows Installer queries the `AdminUISequence` and `AdminExecuteSequence` tables for what actions need to be executed. For the `ADVERTISE` top-level action, the Windows Installer queries only the `AdvtExecuteSequence` table for the actions that need to be run. The `AdvtUISequence` table is empty and is only in the database schema for the purpose of symmetry.

Before the details of the installation process as implemented by the Windows Installer are discussed, we need to take a look at the mechanisms that can be used to launch an installation.

Running the Windows Installer Engine from the Command Line

The Windows Installer engine consists primarily of the `msiexec.exe` and `msi.dll` files, which are installed in the `%SystemRoot%\System32` folder. Which of the three top-level actions the Windows Installer engine executes depends on the switch that is passed to the engine on the command line. For the three top-level actions the applicable command lines are shown below:

INSTALL Top-Level Action:

```
msiexec /i <path to MSI file> <UI Level switch> PROPERTY=value
```

ADMIN Top-Level Action:

```
msiexec /a <path to MSI file> <UI Level switch> PROPERTY=value
```

ADVERTISE Top-Level Action:

```
msiexec /j[m|u] <path to MSI file> <UI Level switch>
```

Note that for the INSTALL, ADMIN, and ADVERTISE top-level actions there is something called the UI Level switch that can be placed on the command line. Also note that you can set the value of public properties on the command line for the INSTALL and ADMIN top-level actions. For the ADVERTISE top-level action, public properties cannot be set on the command line.

The switches used to identify which top-level action the Windows Installer needs to implement are fairly self-explanatory. The only switch that might be confusing is the /j for the ADVERTISE top-level action. Here the /j stands for just-in-time and the optional modifier 'm' stands for advertising to all users of the machine and the 'u' modifier stands for advertising to the current user only.

In the past, installations have either run with a complete set of dialogs that led the end user through the installation process or they have run silently where there is no user interaction possible and no indication that an installation is being run. The Windows Installer introduced four user interface levels instead of just two. The user interface level used for a particular installation is important to how the Windows Installer mechanism processes the MSI file. The following list provides a definition of the four user interface levels.

Full: A full user interface level is where all authored modal and modeless dialogs are displayed. Built-in modal error message dialogs are also displayed. This is the default user interface level and no switch is required for running the installation with this type of user interface level.

An authored dialog is one that is defined in the various database tables that deal with the user interface. A built-in dialog is one that is displayed by the Windows Installer from the dialog templates in msi.dll. A modal dialog is one where user

interaction is required to close the dialog before the installation can continue. A modeless dialog is one where the installation process can continue without requiring the end user to close the dialog.

Reduced: A reduced user interface is one where only authored modeless dialogs are displayed, such as the progress dialog. Built-in modal error message dialogs are also displayed. The UI Level switch to implement this user interface level is /qr. The 'q' stands for quiet and the 'r' stands for reduced.

Basic: A basic user interface level is one where only built-in modeless dialogs are shown. In general this is the built-in version of the progress dialog that is authored in the database. The switch for this user interface level is /qb where the 'b' stands for basic.

None: This is a completely silent installation that displays no dialogs. The switch to obtain this user interface level is /qn.

There is another important command line switch that allows you to create a log file while the installation is run. On the command line after the path to the MSI package, you can use the /l switch followed by some modifiers, which will create a log file that you also have to identify. There are many identifiers that can be used to log various items during the running of the installation but the one that you will use most of the time is the one that lets you log everything and have it logged in verbose mode. A sample command line for implementing this type of logging is as follows:

```
msiexec /i <path to MSI file> /l*v "C:\Temp\install.log"
```

This command line generates a verbose log named install.log in the Temp directory of the C: drive. Creating log files is a useful way to debug installation packages. An installation logs all operations that the Windows Installer performs up to the very point of the installation's failure. Reading the log file is an excellent way to find out where to start looking for the problem.

The above introduction to running the Windows Installer engine from the command line is not complete. For a complete description, see the Windows Installer help and find the first topic in the appendix that is named "Command Line Options". This topic provides a complete description of all the command line options that can be used with the Windows Installer.

You can also double-click on an MSI file in Windows Explorer and launch the installation in that manner. This is possible because the .msi extension is registered to the following command line.

```
"%SystemRoot%\System32\msiexec.exe" /i "%1" %*
```

This is the command for the Open verb under HKEY_CLASSES_ROOT in the registry.

Running the Windows Installer Engine Programmatically

You can create programs that will run installations using the Windows Installer engine. This is how Setup.exe works. The Windows Installer provides a complete set of API functions that programs can access. All these functions are exported by msi.dll. There are three particular functions that we want to take a brief look at here. These are the MsiInstallProduct, the MsiSetInternalUI, and the MsiEnableLog functions.

```
UINT MsiInstallProduct(
    LPCTSTR    szPackagePath,    // path to MSI package
    LPCTSTR    szCommandLine    // command line for package
);
```

This function takes two arguments, one being the path to the MSI file and the other being the command line to be sent to the Windows Installer engine. This command line does not include any switches. It can be only a set of public properties and the values to which they are to be set. How then does the Windows Installer know what type of installation to run? The answer is found in a special public property defined by the Windows Installer. This public property is named ACTION and it takes as a value the name of the top-level action to be run. For example, if the command line argument were initialized to the following string, the Windows Installer would perform a standard installation in response to the call to the MsiInstallProduct function.

```
LPCTSTR    szCommandLine = "ACTION=INSTALL";
```


How do you tell the Windows Installer what user interface level to use for the installation? You make a call to the `MsiSetInternalUI` function to define the level you want before calling the `MsiInstallProduct` function. Finally, how can you have log file created when the installation is run? Before you call the `MsiInstallProduct` function, make a call to the `MsiEnableLog` function to define what you want logged during the installation.

The full details of these functions can be found in the Windows Installer help. The Windows Installer help also describes an automation interface exposed by the Windows Installer that can be used to do the same operations as described above. C++ developers would use the API functions just described and Visual Basic programmers would use the automation interface methods. These functions are introduced here because it provides some background for the discussion in Chapter 4 about the run-time architecture of InstallShield Developer.

The next section discusses what the Windows Installer does with the MSI package after it is passed either on the command line or through a call to the `MsiInstallProduct` function.

The Operations of the INSTALL Top-Level Action

We will use the steps shown in Figure 3-27 to walk through the various operations carried out during a Windows Installer based install. The diagram in Figure 3-27 is applicable to Windows NT 4.0 and Windows 2000. It can be seen that there are two processes shown in this figure. On Windows 9x machines there will be only one process running. The operation on Windows 9x machines differs from that on Windows NT-based machines.

There is always a client process regardless of the operating system on which the installation is running. On Windows NT-based machines, there is also a service process and there must be communication between the client and the service processes. The Windows Installer runs as an NT service as the second process. The service process is necessary on NT-based machines because of the security functionality that is part of these operating systems. Under certain circumstances the service process manipulates the security to allow *elevated privileges*. When granted, elevated privileges means that end users who would not normally be able to install

software will be able to do so. This is a critical functionality on which the Windows 2000 software deployment mechanism depends.

On NT-based operating systems, the actions that are entered in the InstallUISequence table are run in the client process and the actions entered in the InstallExecuteSequence table are run in the service process. This is also applicable to the other sequence tables for the other top-level actions. On Windows 9x machines, there is only a client process and the actions in both the InstallUISequence and InstallExecuteSequence are run in the client process.

On NT-based machines, the client process and the service process communicate with each other through the values of the public properties. Remember public properties are those that have names specified in all upper-case letters. The values of private properties are not shared across the process boundary between the client and the service process. On Windows 9x machines there is no communication requirement because there is only one process, so the values of both public and private properties are available when running the actions in either the InstallUISequence or the InstallExecuteSequence tables.

The following list discusses each of the numbered items shown in Figure 3-27.

Step 1: The previous two sections discussed how the Windows Installer can be launched from the command line, launched programmatically, or launched from Windows Explorer. Regardless of how the installation is launched, the Windows Installer makes a copy of the MSI file, places it in the Temp directory, and gives it a unique name. On Windows 2000 the location of this Temp directory is the following location:

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

The Windows Installer sets the value of the DATABASE public property to the name and location of the MSI file in the Temp directory. This temporary copy of the MSI file is then loaded into memory in the client process.

Step 2: The Windows Installer checks the user interface level that has been specified. If the user interface level is Full or Reduced, it begins with a query of the InstallUISequence table and then proceeds to the InstallExecuteSequence table. If the user interface level is Basic or None, the Windows Installer skips the actions in the InstallUISequence table and queries only for the actions in the

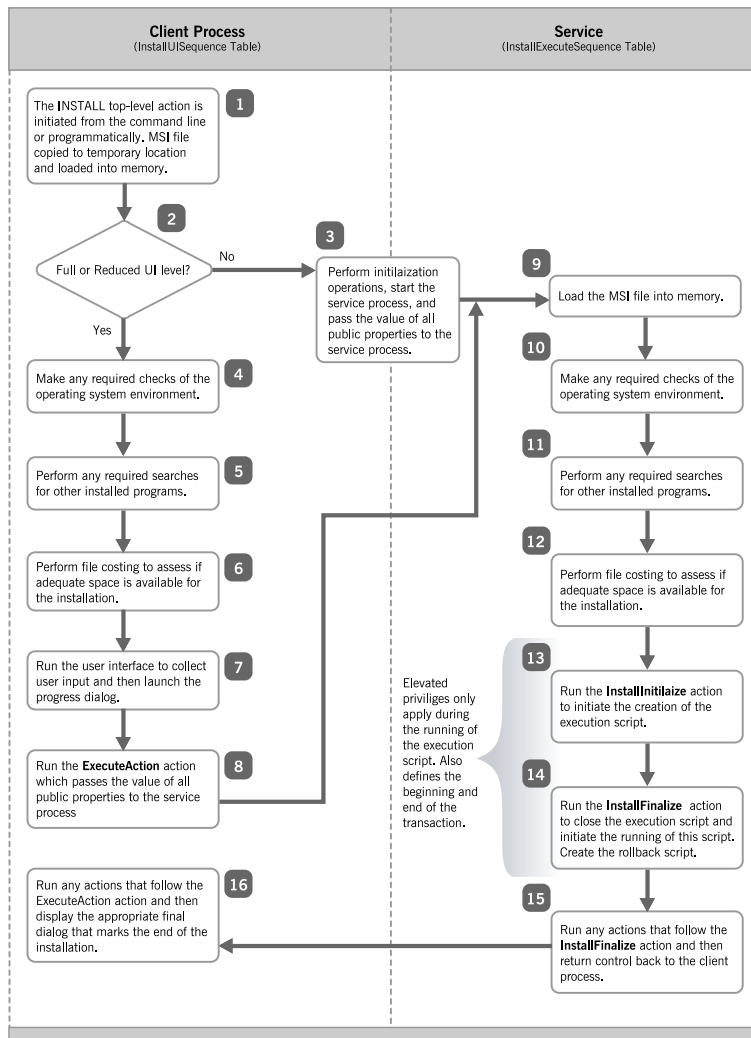


Figure 3-27: *The Windows Installer process for performing an installation on Windows NT 4.0, Windows 2000 and Windows XP.*

InstallExecuteSequence table. Remember that the Basic and the None user interface levels display only built-in dialogs and do not display any of the authored dialogs defined in the database.

Step 3: If the user interface level is either Basic or None, the client process performs some initialization operations and then begins running the service

process. The client process passes to the service process the value of all public properties that have been set on the command line, as well as the value of the DATABASE public property. When control is passed to the service process the database is cached in the following location:

```
%SystemRoot%\Installer
```

The DATABASE public property sends the cached name of the MSI database to the service process.³

Step 4: Many installations should be allowed to run only on certain environments because the application that is being installed will run only on these environments. Therefore at the very beginning of an installation, an action called `LaunchConditions` is run. The `LaunchConditions` action queries the `LaunchCondition` table for any conditions that need to be checked. You can author into this table those conditions that have to be met before the installation can be allowed to continue. If any of the conditions is not met (returns `FALSE`), the Windows Installer terminates the installation by first displaying a message box with a message from the `LaunchCondition` table. When the end user clicks the OK button on the message box, the installation ends.

Step 5: After checking the environment, you may then want your installation to search the target system to see if any required applications are installed. It may also be the case that the version of the product being installed is a competitive upgrade that was sold to the end user on the assumption that he or she owned the competing product. It is then necessary to search for this competing product before allowing the installation to proceed. There are three actions that are used to implement the various search mechanisms and there are six tables that these actions use to perform the search. A successful search sets a property that can be used in controlling other actions. Conversely, the property is not set if the search fails.

Step 6: After making all the necessary checks to validate that the installation can continue, the Windows Installer performs a set of actions that check if the amount of space on the target system is adequate for the default set of features that will be installed. This set of actions is called file costing and a number of important operations are carried out as part of these actions. File costing takes into account the size difference between the files that are being copied and the same files that may be on the target system. File costing also calculates the

additional space required by the registry, new initialization files created, entries into existing initialization files, and new shortcuts being added to the system. As part of file costing, the Windows Installer has to perform three operations. These three operations are listed and discussed below:

- ***Evaluate Feature Conditions:*** The Level attribute specified in the Feature table has already been discussed earlier in this chapter. The initial value for the Level attribute can be modified at install time based on the truth of condition. The conditions for modifying the Level value for a feature are entered into the Condition table. The Condition table has three columns, a foreign key into the Feature table, a new value for the Level attribute for the feature, and a condition. If a condition in this table evaluates to TRUE then the Level attribute for the feature will be changed to the value specified in the Condition table. This mechanism allows a small amount of control over what features get installed by default on a particular system. It is only logical that this has to be evaluated as part of the file costing mechanism.
- ***Evaluate Component Conditions:*** As already mentioned, a condition can be placed on any component. The Component table has a column for defining these conditions. If the condition evaluates to TRUE, the size of the component is included as part of the file costing calculation. Otherwise, the size of the component is not considered.
- ***Resolve Directory Table:*** Each row in the Directory table indicates a directory path for both the source and the target. When the entries in the first column of the Directory table are resolved during file costing, the rows in the Directory table become properties in the in-memory version of the Property table. As already stated directory path properties always have an ending backslash. After file costing is complete, you can query the Property table to access these locations.

The Directory column in the Directory table contains identifiers that get resolved to absolute locations on both source and target locations. The Directory_Parent column is a foreign key back into the Directory column of the same table. As seen in Figure 3-28 the first row in the Directory table does not have an entry in the Directory_Parent column and only one row with this column having a Null

value is allowed. This tells the Windows Installer that this row defines the root target and root source directories. It is required that the TARGETDIR property be used in the first column of the first row of this table. It is also necessary that the SourceDir property be used in the DefaultDir column of this same row. The TARGETDIR property represents the root location of the installation and normally would get set at the command line. In the implementation by InstallShield Developer, as you will see, this property is not used. InstallShield Developer uses the INSTALLDIR property as the root target location for the installation. The SourceDir property is set by the Windows Installer to be the path to the MSI file on the source media.

The entry in the DefaultDir column is defined to be a sub-folder under the folder defined in the Directory_Parent column. The general format of the DefaultDir column in the Directory table is as follows:

```
[target location]:[source location]
```

As shown, there can be different locations for the target and the source if a colon (:) separates them. You define the target and the source locations using the short and long file naming conventions with a pipe symbol (|) as a separator between the two conventions. If either the target location or the source location is replaced with a period (.), this means that there is no sub-folder defined under the entry shown in the Directory_Parent column. In Figure 3-28 there are many entries in the DefaultDir column as follows:

```
.: PROGRA~1 | program files
```

Tables	Directory	Directory_Parent	DefaultDir
Directory	TARGETDIR		SourceDir
DirLocator	ISCommonFilesFolder	CommonFilesFolder	Instal~1\InstallShield
DuplicateFile	INSTALLDIR	DEVELOPER_ART_BASIC	.
Environment	DEVELOPER_ART_BASIC	INSTALLSHIELD	DEVELO~1\Developer Art Basic
Error	ISUpdateServiceFolder	ISCommonFilesFolder	UPDATE~1\UpdateService
EventMapping	ISYourProductDir	ISYourCompanyDir	YOURPR~1\Your Product Name
Extension	INSTALLSHIELD	ProgramFilesFolder	INSTAL~1\InstallShield
Feature	ISYourCompanyDir	ProgramFilesFolder	YOURCO~1\Your Company Name
FeatureComponents	AdminToolsFolder	TARGETDIR	..Admin~1\AdminTools
File	AppDataFolder	TARGETDIR	..APPLIC~1\Application Data
FileSFPCatalog	CommonAppDataFolder	TARGETDIR	..Common~1\CommonAppData
Font	CommonFiles64Folder	TARGETDIR	..Common64
Icon	CommonFilesFolder	TARGETDIR	..Common
IniFile	DesktopFolder	TARGETDIR	..Desktop
IniLocator	FavoritesFolder	TARGETDIR	..FAVORI~1\Favorites
InstallExecuteSequence	FontsFolder	TARGETDIR	..Fonts
InstallUISequence	GlobalAssemblyCache	TARGETDIR	..Global~1\GlobalAssemblyCache
IsolatedComponent	LocalAppDataFolder	TARGETDIR	..LocalA~1\LocalAppData
LaunchCondition	MyPicturesFolder	TARGETDIR	..MyPict~1\MyPictures
ListBox	PersonalFolder	TARGETDIR	..Personal
ListView	PrimaryVolumePath	TARGETDIR	..Primar~1\PrimaryVolumePath
LockPermissions	ProgramFiles64Folder	TARGETDIR	..Prog64~1\Program Files 64
MIME	ProgramFilesFolder	TARGETDIR	..PROGRA~1\program files
Media	ProgramMenuFolder	TARGETDIR	..Programs
MoveFile	SendToFolder	TARGETDIR	..SendTo
MsiAssembly	StartMenuFolder	TARGETDIR	..STARTM~1\Start Menu
MsiAssemblyName	StartupFolder	TARGETDIR	..Startup
MsiDigitalCertificate	System16Folder	TARGETDIR	..System
MsiDigitalSignature	Svstem64Folder	TARGETDIR	..Svstem64

Tables: 84 Directory - 34 rows No column is selected.

Figure 3-28: The Directory table from the Developer Art installation database.

This type of entry indicates that there is no sub-folder for the target location but that there is a sub-folder called "program files" under the source location. When there is only a period (.) in the DefaultDir column it means neither the target nor the source location have any sub-folders defined. When there is only one entry for a sub-folder without a colon (:), this means that the sub-folder name is the same for both target and source locations.

Let's first look at how the target paths are resolved and then at how the source locations are resolved.

The first column of the Directory table contains a number of entries such as ProgramFilesFolder. These are the names of properties that are defined by the operating system. When the Windows Installer first launches, it sets the value of all these properties to their absolute location on the target system. When the Windows Installer starts to resolve the Directory table for the target locations and

it finds an entry in the first column that is already defined by a property, it stops target resolution of the identifier because the identifier already points at an absolute path on the target system.

However, when the Windows Installer comes across an identifier in the first column such as `INSTALLDIR` and it does not find this defined in the Property table, it has to go further in order to resolve the absolute location to which this identifier is pointing. As shown in Figure 3-28, the Windows Installer will find that the `INSTALLDIR` identifier points at a location defined by the `DEVELOPER_ART_BASIC` identifier and that this location has no sub-folder as indicated by the period (`.`) in the `DefaultDir` column. The `DEVELOPER_ART_BASIC` identifier is a key into the first column of the Directory table.

The Windows Installer finds the `DEVELOPER_ART_BASIC` identifier in the first column and if it does not know the location to which the `DEVELOPER_ART_BASIC` identifier is pointing, it starts to resolve this location. It finds that the `DEVELOPER_ART_BASIC` identifier is a location pointed to by the `INSTALLSHIELD` identifier with a sub-folder named "Developer Art Basic". The `INSTALLSHIELD` identifier is another key into the first column of the Directory table. Once again the Windows Installer needs to resolve the location at which the `INSTALLSHIELD` identifier is pointing. This identifier is found to point at the location defined by the `ProgramFilesFolder` identifier with a sub-folder named "InstallShield". Now all the pieces are in place to find out where these other identifiers are pointing. On a typical system the resolution would be as follows:

```

ProgramFilesFolder:    C:\Program Files\
INSTALLSHIELD:        C:\Program Files\InstallShield\
DEVELOPER_ART_BASIC:  C:\Program Files\InstallShield\
                        Developer Art Basic\
INSTALLDIR:           C:\Program Files\InstallShield\
                        Developer Art Basic\

```

At the completion of file costing, all the identifiers that do not already have values in the in-memory version of the Property table are made into properties and the values of these properties are absolute target locations. These values always have an ending backslash. In other words, until file costing is complete, the value of the `INSTALLDIR` property cannot be accessed because it does not yet exist as a

property. After file costing, the value of a property named `INSTALLDIR` is accessible.

Next, we will discuss the resolution for defining the source locations on the distribution media. The resolution process is much the same except now all identifiers in the `Directory` column of the `Directory` table need to be resolved since these identifiers cannot be set without knowing where the media source is located. That information is provided by the value of the `SourceDir` property. Lets start with the resolution of the `ProgramFilesFolder` identifier.

For the `ProgramFilesFolder` identifier, the value in the `Directory_Parent` column, `TARGETDIR`, is a key into the first row of the `Directory` table. Since the Windows Installer is now resolving the source locations, the `ProgramFilesFolder` identifier points to the following location.

```
[SourceDir]program files
```

The square brackets around the `SourceDir` property name indicate that the property name and the square brackets are to be replaced with the actual value of the `SourceDir` property. There is no backslash between `[SourceDir]` and "program files" because the `SourceDir` property already has an ending backslash.

By taking this process to its logical end, the `INSTALLSHIELD` identifier resolves to the following source location:

```
[SourceDir]program files\InstallShield\
```

The `DEVELOPER_ART_BASIC` identifier resolves to the following location:

```
[SourceDir]program files\InstallShield\Developer Art Basic\
```

The `INSTALLDIR` identifier resolves to the same location because there is no sub-folder shown in the `DefaultDir` column. Since this column cannot be `Null`, the period is necessary.

Step 7: After file costing is complete, the Windows Installer runs the appropriate user interface based on what type of installation is being run. The types of installations that can be run when the `INSTALL` top-level action has been specified are a fresh install, a maintenance install, a patch install, or resumed install. The type of install being performed determines the user interface that is

displayed to the end user. For a fresh install, the user interface guides the end user through a number of wizard dialogs and asks various questions, such as where the application should be installed and what type of setup the user wants. The user interface for the maintenance type of installation asks the end user what type of maintenance to perform on the already installed application. This can be a modification of the features presently installed, a repair of the installation, or a removal of the application from the system. The user interface for the other two types of installations normally just warn the user what type of install is taking place and then the Windows Installer runs the installation. Just prior to the launching of the installation a progress dialog is displayed on the screen and it stays on the screen through the full installation process.

Step 8: Once the user interface has displayed the progress dialog on the screen, the installation moves on because the progress dialog is a modeless dialog. This is where the client process launches the service process on NT-based machines. Calling the `ExecuteAction` action in the `InstallUISequence` table implements the running of the service process. This action initiates the processing of the `InstallExecuteSequence` table by the service process. The `ExecuteAction` action sends to the service process the value of all public properties that have been defined in the client process. It does not have to send the value of those public properties that are built into the database if they have not changed. This is because the service process will have access to those values when it opens the database in its own process space.

On Windows 9x machines, there is no second process so there is no need to transfer the values of any properties. The processing of the `InstallExecuteSequence` table actions is performed in the client process, which has access to all the properties in the property table as well as the in-memory version of the property table.

Step 9: Here, the Windows Installer is in the service process and the first thing that this process needs to do is load the MSI file into memory. It loads the MSI file from the location as defined by the `DATABASE` public property. As already stated under Step 3 this location is where the database is cached. This location is `%SystemRoot%\Installer`. When the MSI file is loaded, the service process queries the `InstallExecuteSequence` table for the actions that are to be run.

Step 10: This operation is the same as described under item 4. This same check for the environment is necessary in both the `InstallUISequence` and `InstallExecuteSequence` tables because of the possibility of the end user running a user interface level of Basic or None. In this case the check defined in the `InstallUISequence` table would be skipped.

Step 11: This operation is the same as described under item 5. This same check for installed applications is necessary in both the `InstallUISequence` and `InstallExecuteSequence` tables because of the possibility of the end user running a user interface level of Basic or None. In this case the check defined in the `InstallUISequence` table would be skipped.

Step 12: The file costing performed in the service process on an NT-based system makes any corrections to the file costing performed in the client process based on whether there have been any changes in the default set of features to be installed. Changes after the original file costing can occur if the end user performs a custom setup and changes the default set of features to be installed. If the default list of features does not change, additional file costing is not necessary. If the installation is run with a Basic or None user interface level, the full file costing is performed in the service process. Resolution of the `Directory` table is performed again but, because the resolution of identifiers such as `INSTALLDIR` was done in the client process and now is a member of the in-memory `Property` table, the target side of the resolution process is not necessary.

On Windows 9x machines, the file costing is done only once. If the user interface level is Full or Reduced, the file costing is performed in the `InstallUISequence` table. If the user interface level is Basic or None, the file costing is performed in the `InstallExecuteSequence` table. This is because the file costing operation can only be performed once per process. The problem that this presents is that if the end user makes changes to the selection of features in the custom setup dialog then the file costing for the application will not be accurate.

Step 13: The `InstallInitialize` action defines the beginning of those actions that make actual changes to the target system. Up to this point all actions have been doing nothing but collecting information. From this point to where the `InstallFinalize` action is called, actions that are going to make changes to the system are not executed. Instead the actions are written into an execution script that the Windows Installer is creating in a hidden directory. The important item to remember about this is that if an action is not written into the execution script,

it will not receive elevated privileges in a Windows 2000 network that is implementing software deployment from a central point using the Group Policy Editor (GPE).

In the `InstallExecuteSequence` table for the Developer Art installation, there are a large number of actions that occur after the `InstallInitialize` action and before the `InstallFinalize` action. Any action that is to make changes to the system will be written into the execution script. An action that does not have any data in the requisite table or tables will not be making changes to the system, so it will not be written into the execution script.

Step 14: The `InstallFinalize` action closes out the creation of the execution script and initiates the running of this script. As each line in the script is executed it is written into a rollback script that would be used if the end user either cancels the installation or there is an installation error. The rollback script is used to reverse any changes that may have been made to the target system and is the key to the transactional nature of a Windows Installer-based install. The completion of this action ends the transaction that was started by the `InstallInitialize` action.

Step 15: After the `InstallFinalize` action there can be additional actions if you want to do additional things at this point. These are run at this time and when these additional actions are executed, the service process returns control to the client process. Actions placed after the `InstallFinalize` action should not be designed for making changes to the system since any such changes cannot be rolled back in case of a failure of the installation.

Step 16: Once control is returned to the client process, any actions in the `InstallUISequence` table placed after the `ExecuteAction` action are run. You can consider the `ExecuteAction` like a function call because, after control returns to the client process, execution continues with the action following the `ExecuteAction` action. When all actions after the `ExecuteAction` action are executed, if any, the Windows Installer displays a finish dialog informing the end user that the installation has completed successfully.

The dialogs displayed at the end of installation, whether it is successful or not, have to have special sequence numbers in the `InstallUISequence` table. These special sequence numbers and their meaning are shown below:

Sequence #	Description
-1	Informs the end user that the installation has completed successfully.
-2	Informs the end user that the end user canceled the installation.
-3	Informs the end user that there was a fatal error during the installation.

If the installation is not successful the database file that was cached in %SystemRoot%\Installer is deleted. The cached database is only left in this location if the installation is successful.

This cached file is sometimes referred to as the local package and is used to perform maintenance operations on the installed applications. In this same location the Windows Installer also caches the resource files used to supply icons for advertised shortcuts. The cached resource files will be in a sub-folder that uses the ProductCode property as the sub-folder name. The ProductCode property is a GUID.

Extending the Windows Installer Functionality

The Windows Installer provides many standard actions that are sufficient to execute many installation operations. However, there are situations where you might find that the standard actions do not perform what needs to be accomplished during a particular installation. To accommodate the fact that there will never be enough standard actions to satisfy all needs, the Windows Installer provides a mechanism to extend its built-in functionality through the use of custom actions. There are numerous reasons that the creation of a custom action may be necessary, from being able to validate the input of a serial number to the running of a child third party installation. This chapter introduces the concept of custom actions. The creation and use of InstallScript custom actions is discussed in Chapter 11.

Custom Action Categories

There are two major categories of custom actions: immediate custom actions and deferred custom actions. The deferred category of custom action has four sub-categories, which are install, rollback, commit, and system context. More specifically, the system context sub-category is a modifier for the install, rollback, and commit custom actions. These categories are defined in the following list:

Immediate custom actions: Custom actions are inserted into the sequence tables just like standard actions. When the Windows Installer encounters an immediate custom action during its query of one of the sequence tables, it executes the custom action immediately. Immediate custom actions are used to query the system, set property values, and add temporary rows to the database tables. Immediate custom actions are not used to make changes to the system because they would not receive elevated privileges in the case of a locked down environment.

Immediate custom actions can be placed almost anywhere in the sequence tables used by the INSTALL or the ADMIN top-level actions. No custom actions of any kind can be used in either the AdvtUISequence or AdvtExecuteSequence tables. There are a few restrictions about where they can be placed depending on the type of custom action being implemented. Immediate custom actions have the same privilege level as the user who is signed on to the machine.

Deferred custom actions: A deferred custom action is first written into the execution script and then, when the execution script is run, the deferred custom action is run. This means that a deferred custom action can be placed only in those tables where the execution script is generated. These two tables are the InstallExecuteSequence and the AdminExecuteSequence tables. In these two tables, deferred custom actions have to be placed between the InstallInitialize and the InstallFinalize standard actions. Remember that it is only the actions between these two standard actions that are written into the execution script.

As mentioned above there are four sub-categories of deferred custom actions. These sub-categories are defined in the following list.

- **Install:** An install deferred custom action is one that runs during the performance of a normal installation or uninstallation. This normal

installation can be a fresh install, a maintenance install, a patch install, or a resumed install. It can also be an administrative installation. Even if elevated privileges have been granted to the user this sub-category of deferred custom action will still only run with the same privilege level as the user signed on to the machine.

- ***Rollback:*** A rollback custom action is not executed when the execution script is run. It is written into the rollback script and executed only if the rollback script is run. The rollback script is run only if the user cancels the installation or there is a Windows Installer error. The importance of rollback custom actions is to undo any changes that may have been made to the target system by an install custom action prior to the rollback operation. This type of custom action will run with the some privilege level as the user signed on to the machine regardless of whether elevated privileges have been granted.
- ***Commit:*** A commit custom actions runs at the completion of a successful installation. This makes it the complement of a rollback custom action, which runs when an installation does not complete successfully. Commit custom actions are also written into the rollback script and are executed from that script when the InstallFinalize action indicates a successful running of the execution script. It is possible to disable rollback on a system that has limited space for installing an application and if this is done then any commit custom actions are also affected. If no rollback script is created because of disabling rollback, no commit custom actions will be able to run. This sub-category of custom also runs only with the same privilege level as the user signed on to the machine.
- ***System context:*** The system context sub-category is a modifier for the previously described sub-categories of deferred custom actions. Using the system context modifier allows the install, rollback, and commit custom action sub-categories to have local system account privileges. In technical terms this means that the modified custom actions do not impersonate the user signed on to the machine. Local system account privileges are the same as having administrative privileges on the local machine. These local system account privileges

are only available, however, if elevated privileges have been granted by one of the mechanisms available in a Windows 2000 network or through the setting of the appropriate per-user and per-machine policies in the registry. If elevated privileges have not been granted then the use of the system context modifier has no effect and the custom actions will still only run with the same privilege level as the user signed onto the machine.

Lets now look at the types of custom actions that can be created. When we talk about the types of custom actions we are referring to the implementation details of creating custom actions.

The Types of Custom Actions

Custom actions can be divided into eight different types. Of these types, five require some programming expertise and the other three can be created without any programming. The description of these eight types of custom actions is given below:

Executables: A custom action can launch an executable and this executable can be installed with the application, be included as a stream in the Binary table, or it can already exist on the target machine. An executable custom action can be configured to run synchronously or asynchronously. If the custom action runs asynchronously then it is possible to specify that the Windows Installer is to wait at the end of processing the sequence table in which the custom action has been inserted for the custom action to complete. It is also possible to specify that the custom action will continue executing even after the installation has completed.

Dynamic Link libraries: A DLL type of custom action runs as a separate thread in the same process as the Windows Installer, (either the client process or the service process). The DLL that exports the functionality for the custom action can only be installed with the application or it can be contained as a stream in the Binary table. You cannot directly call a function in a DLL that is already on the target system as a custom action. As with an executable custom action it can run synchronously or asynchronously, but a DLL custom action cannot continue after the installation is complete because the process that is running it has terminated. One of the benefits of a custom action exported from a DLL is that it has access to the session handle and thus is able to access the database at run time.

InstallScript: InstallScript can be used to implement custom actions. As already discussed, InstallScript is the scripting language that was developed by InstallShield Software Corporation. The Windows Installer does not know anything about InstallScript, so what is running in the background is a DLL that runs the scripting engine. The only thing that the Windows Installer sees is a DLL custom action. InstallScript custom actions are covered in Chapter 11.

VBScript: A VBScript custom action can be in a .vbs file, or the script code itself can be included in either the Property table or in the CustomAction table. VBScript custom actions implemented in a file can either be installed with the application or streamed into the Binary table. As with an executable custom action, a VBScript custom action can run synchronously or asynchronously. The Windows Installer is the host for script-based custom actions and it is not necessary to have the Windows Scripting host installed on the target machine. The only thing that needs to be installed is the system file scrrun.dll.

JScript: A JScript custom action can be either in a .js file or the script code itself can be included in either the Property table or in the CustomAction table. JScript custom actions implemented in a file can either be installed with the application or streamed into the Binary table. As with an executable custom action, a JScript custom action can run synchronously or asynchronously. The Windows Installer is the host for script-based custom actions and it is not necessary to have the Windows Scripting host installed on the target machine. The only thing that needs to be installed is the system file scrrun.dll.

Formatted Text: A formatted text custom action is used to set a property or an install directory from a formatted text string. A formatted text string is one that is processed to resolve embedded property names, table keys, environment variable references, and other special sub-strings. The embedded property names, table keys, and environment variable references are surrounded by square brackets ([]). This type of custom action can be defined only as immediate and it runs only synchronously.

Error: The error type of custom action displays a specified error message and returns failure, terminating the installation. The error message to be displayed can be supplied as a string or as an index into the Error table. The key to using this type of custom action is to condition it so that it runs only when it should. Once it runs, the installation must terminate. This type of custom action runs only synchronously.

Nested Install: Using a nested install custom action is the only means to enable a child install that is also defined in an MSI package. The basic mechanism of the Windows Installer will not allow two different installations to enter the service process at the same time. The nested install custom action was implemented to get around this particular functionality. A nested install custom action can run only synchronously and it cannot be a deferred custom action. A nested install shares the same user interface level and the same logging settings as the parent install.

The Other Types of Windows Installer Packages

The Windows Installer recognizes three additional types of packages. All of these packages are COM Structured Storage files of one type or another. These packages are merge modules, transforms, and patch packages. Only the merge module can be opened with the Orca database-editing tool. Let's take a brief look at these other Windows Installer packages.

Merge Modules

It is possible to combine two installer databases by merging them together using one of the tools that come with the Windows Installer SDK. Merging two databases, however, has certain problems. You cannot merge two databases if the schemas of the two databases are different. Even if the schemas of the two databases are the same, there is the possibility for a row merge conflict. A row merge conflict occurs when the same table in both databases has a row that has the same primary key but different data. When there is a row-merge conflict, the merging of the two databases proceeds, but the conflicts are reported in another table that is created for that particular purpose.

You can create a special type of database, called a merge module, which gets around the merging problem described above. A merge module is a simplified form of a Windows Installer installation package. Merge modules cannot be installed separately and are only a build-time entity. After a merge module is merged with a main

installation database, the merge module is no longer required for the installation to proceed.

A merge module contains a relational database, summary information stream, and a cabinet file stored as a stream. This is just like the Windows Installer packages we have been discussing. Each merge module has a unique identifier that is a GUID. The GUID that uniquely identifies a merge module is also used to create unique names for the primary keys in the tables of the relational database. This is done to avoid the creation of row-merge conflicts that can occur when two databases are merged.

Merge modules are primarily used to package components that can be shared between by many different applications. In fact a merge module cannot define a feature but can define only components. Merge modules provide a great benefit to teams that work on different parts of an application in different parts of the country or the world. Merge modules also allow for third parties to create redistributable components that other software developers can easily include in their applications. Remember that one of the rules for creating components is not to create a component for a file that is already available as a merge module.

The best approach is to create merge modules for all components that might be shared across more than one application. By creating components in merge modules, there is a much better chance of maintaining the correct component codes for components. Breaking the rules of component creation can have unexpected and unpleasant consequences. Chapter 14 covers the creation of a simple merge module.

Transforms

A transform is a Windows Installer file that describes the differences between two installer databases. A transform can add or replace information in the target database. A transform can be applied at run time to the installation database and essentially change the package only in memory. A transform can also be applied at build time, but using a transform in this mode will permanently change the target database.

A typical use of a build-time use of a transform is to create a localized version of an upgraded application. This can be accomplished by first creating a transform of the differences between a base language product and its upgrade. Then this transform is

applied to the various language-specific versions of the product in order to obtain their upgraded versions.

An example of applying a transform at install time is where the LAN administrator wants to limit the set of features that are available to the person installing an application. The LAN administrator creates a temporary version of the database that has only the desired features in it and then creates a transform between this temporary database file and the original database. When the installation of the application is run, that transform is applied to modify the database in memory and the end user has access to only the permitted feature set.

It is important to note that a transform is the difference between two databases. It does not include the Summary Information Stream and it does not include any differences between the files that make up two different installation packages. Transforms cannot be authored. To obtain a transform, you need two different installation packages, the *before* and the *after*. You cannot look directly at a transform with Orca or any other tool. To see contents of a transform, you need to apply it to a database and then view one of the temporary tables that are created. This temporary table is the `_TransformView` table.

Patch Packages

A patch package can be considered a super-charged transform. This type of file contains not only the differences between the databases of two or more installation packages, but also the differences between the files that make up two or more Windows Installer installation packages. A minimal patch package contains two transforms, a Summary Information Stream, and a cabinet file. Patch packages are used to provide the Windows Installer service with a mechanism for implementing updates and upgrades of installed applications. The installed applications need to have been originally installed using the Windows Installer.

Conclusion

This chapter has covered a lot of ground about how the Windows Installer works. To discuss the Windows Installer, we used the Basic MSI project that was used to create the installation package for the Developer Art application. Understanding how the

Windows Installer works is important because even the Standard projects use the Windows Installer to make the changes to the target system. You will probably find it valuable to reread this chapter periodically as you work through the examples presented in the chapters in this book.

The main point of the discussion in this chapter is that the Windows Installer needs the information required for installing an application described in the tables of a relational database. The main elements used to describe an application are features and components. Features are the logical description of the functionality of an application and components provide the actual functionality. Components are installed only when the associated feature is selected for installation.

There is a major difference in how the Windows Installer works on Windows NT/2000 versus how it works on Windows 9x machines. On NT-based machines, there are two processes that run—the client process and the service process. The job of the client process is to run the user interface for the installation. The job of the service process is to make the changes to the target system. The reason that an NT service is used to make the changes to the target system is because an NT service can manipulate the security mechanism on NT-based machines. Being able to impersonate the local system account permits the granting of elevated privileges to users who normally would not have the privileges to perform an installation.

Custom actions are the means provided to setup developers for extending the built-in capabilities of the Windows Installer. The approach used by InstallShield Developer to implement the standard project approach to installation program development is an extensive use of custom actions. To fully understand the run-time architecture of InstallShield Developer, covered in the Chapter 4, an understanding of custom actions is necessary.

Chapter

4

The InstallShield Developer Run-Time Architecture

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

This chapter looks at how InstallShield Developer runs both installation types. We look at the run-time functionality for the typical installation types with emphasis on

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

Fresh Install Run-Time Architecture

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

Fresh Install Using a Standard Project

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

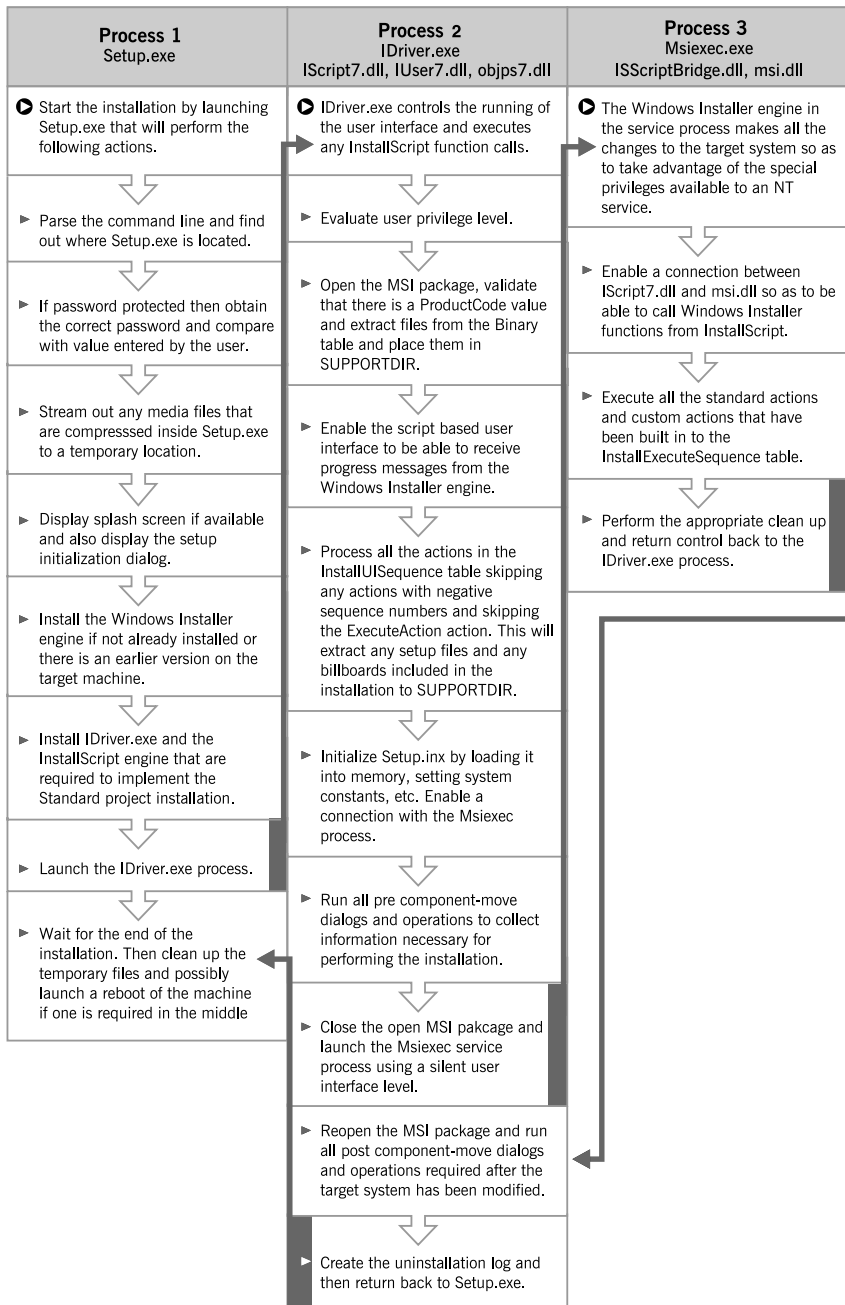


Figure 4-1: Standard project fresh install run-time architecture on Windows NT/2000/XP.

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

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

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

Setup.exe

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

SETUP.INI

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

```

Figure 4-2: *Continued.*

```

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

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

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

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

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

```

Figure 4-2: *Continued*

```

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

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

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

```

Figure 4-2: *Continued.*

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

```

[KEY]
Password=1953684598

```

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

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

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

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

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

COMPRESSED MEDIA FILES

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

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

environment variables are defined. Typically on Windows 2000 this temporary location is as follows, where XXX is a hexadecimal number:

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

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

INITIALIZATION

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

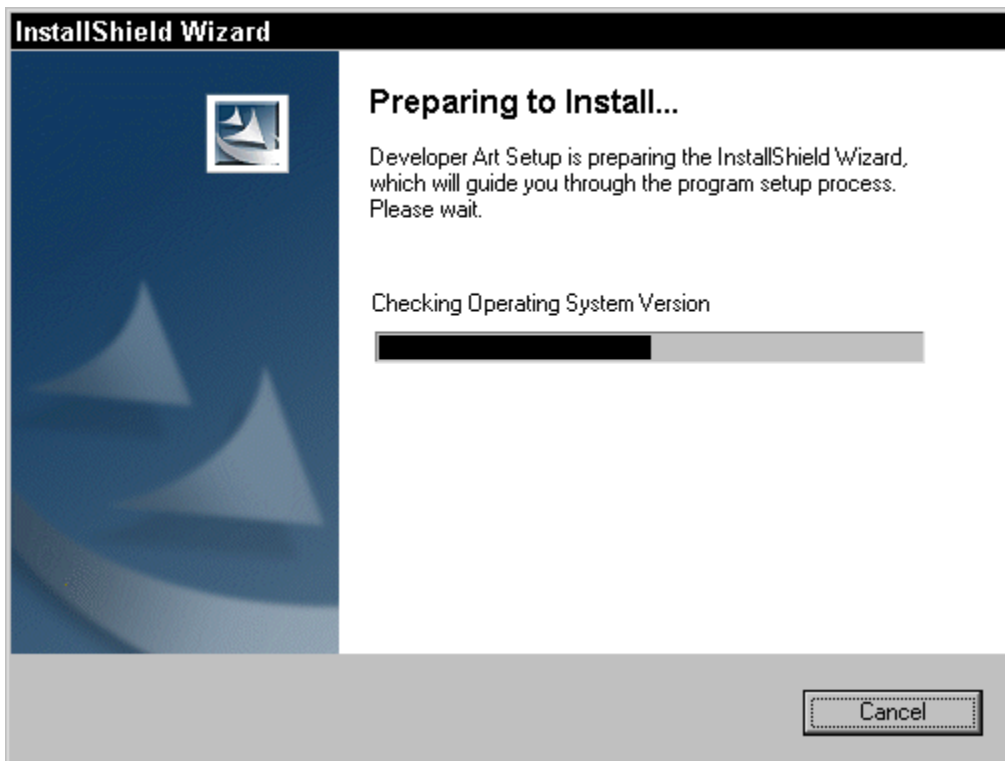


Figure 4-3: *Large initialization dialog displayed by Setup.exe*

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

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

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

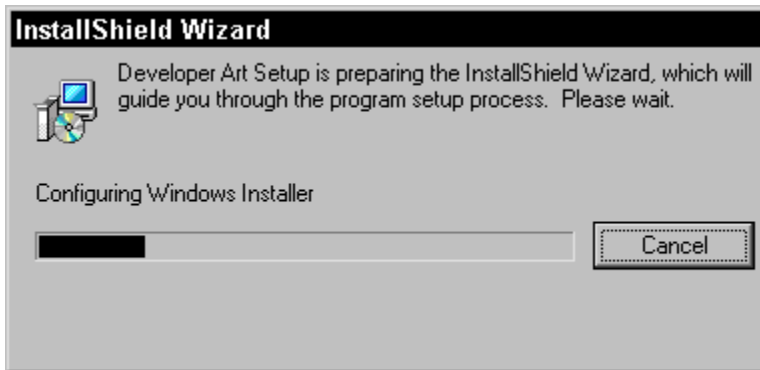


Figure 4-4: *Initialization dialog used when splash screen is displayed.*

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

The large initialization dialog or the small initialization dialog in conjunction with a splash screen are displayed when an end user runs an installation for a product that is already installed. This triggers the maintenance mode. If maintenance mode is launched from the Add/Remove Programs applet, neither of these initialization

dialogs nor the splash screen is displayed. An initialization dialog launched from IDriver.exe is displayed instead.

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

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

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

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

IDriver.exe

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

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

INITIALIZATION

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

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

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

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

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

Tables	Name	Data
Binary	InstallScript	[Binary Data]
BindImage	IsConfig.INI	[Binary Data]
CCPSearch	ISScriptBridge.dll	[Binary Data]
CheckBox	String1033.txt	[Binary Data]
Class		
ComboBox		
CompLocator		
Complus		

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

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

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

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

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

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

Three of the four files in the Binary table are support files and they are streamed out to a temporary folder that is created as part of this operation. The temporary location used is specified by the `TMP` or the `TEMP` environment variable. If neither of these values exists then the temporary location is the Windows folder on Windows NT,

Windows 2000, or Windows XP. For Windows 9.x machines the current directory is used if neither these two environment variables are defined. Typically on Windows 2000 this temporary location is as follows:

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

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

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

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

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

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

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

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

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

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

PROGRAM BLOCK EXECUTION

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

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

Component-move operations: This section of the program...endprogram block launches msixec.exe in silent mode and initiates the running of the actions

in the `InstallExecuteSequence` table. This is covered in the section entitled `Msiexec.exe`.

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

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

UNINSTALLATION LOG

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

When you let the Windows Installer engine make all the changes to the target system, there are a total of three registry entries that are logged in `Setup.ilg`. The first of these entries is the information provided under the following key. This information is used

by the Add/Remove Programs applet. The key shown here is for the Developer Art installation that was created in Chapter 2.

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

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

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

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

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

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

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

There are four system variables that hold values that are written to the log file. You can use three of these system variables in your script to modify the default values that

are written to the log file and to the registry. These system variables are discussed in the following list:

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

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

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

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

```
Developer Art Standard
```

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

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

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

Msiexec.exe

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

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

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

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

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

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

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

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

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

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

This completes the overview of the fresh install run-time architecture of a Standard project. We have gone into some detail here to provide a basis for understanding the run-time architecture of other installation modes. Many of the mechanisms already

described will apply. The next section discusses how the run-time architecture of a fresh install of a Basic MSI project differs from what we have just covered.

Fresh Install Using a Basic MSI Project

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

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

Basic MSI Project With InstallScript Custom Actions

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

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

SETUP.EXE

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

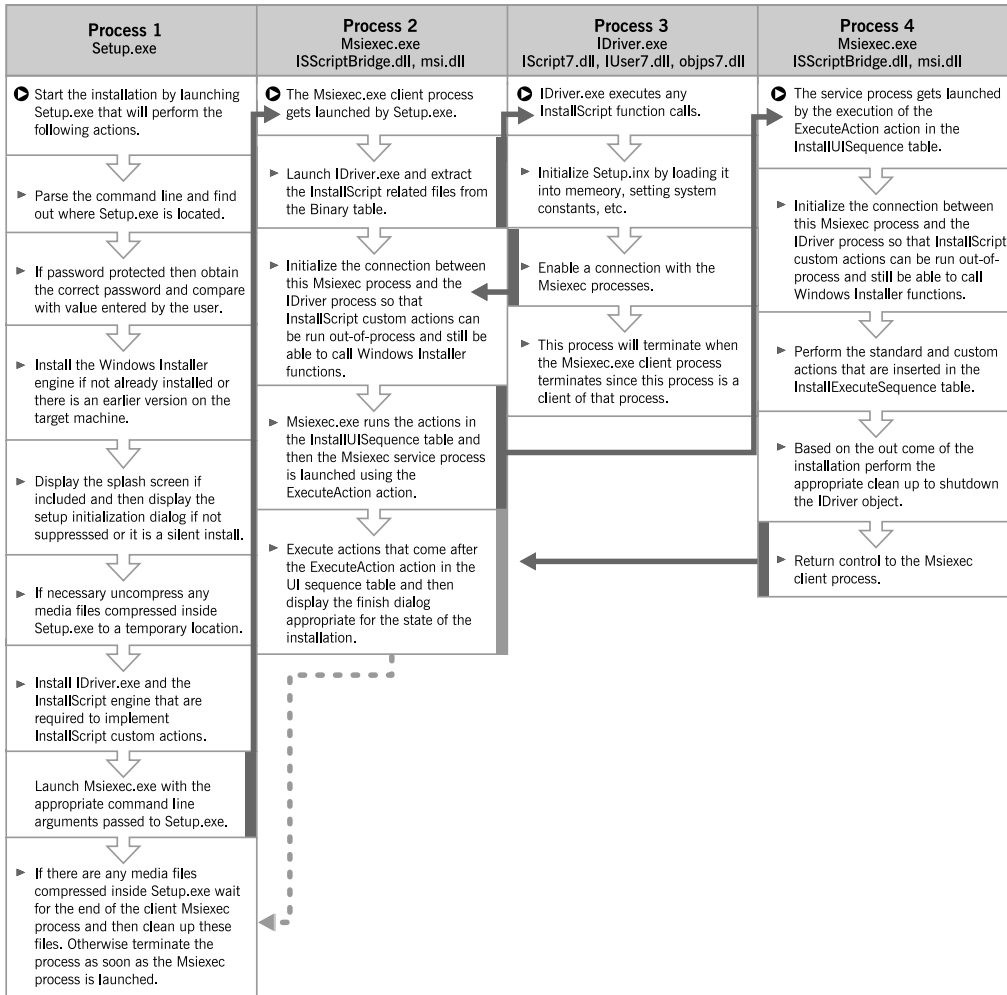


Figure 4-7: Basic MSI project run-time architecture on Windows NT/2000/XP for a fresh install using InstallScript custom actions.

MSIEXEC.EXE – CLIENT PROCESS

The operation of msiexec.exe in the client process is the standard Windows Installer approach, as described in Chapter 3. The standard actions, custom actions, and dialogs that are inserted in the InstallUISequence table are executed in ascending order of the positive sequence numbers that were assigned during the installation package’s build. The first action in the UI sequence table is the ISMsiServerStartup

custom action, with a sequence number of 1. This is a custom action that is implemented in ISScriptBridge.dll and has as its main responsibility the starting of the IDriver.exe process, the extraction of the InstallScript related files from the Binary table, and the initiation of a connection with this process. The script-related files that are streamed in the Binary table are streamed out to a temporary location. This location is set as the value of the SUPPORTDIR InstallScript system variable. This is the same location as described for the Standard project install.

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

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

IDRIVER.EXE

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

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

MSIEXEC.EXE – SERVICE PROCESS

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

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

Basic MSI Project Without InstallScript Custom Actions

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

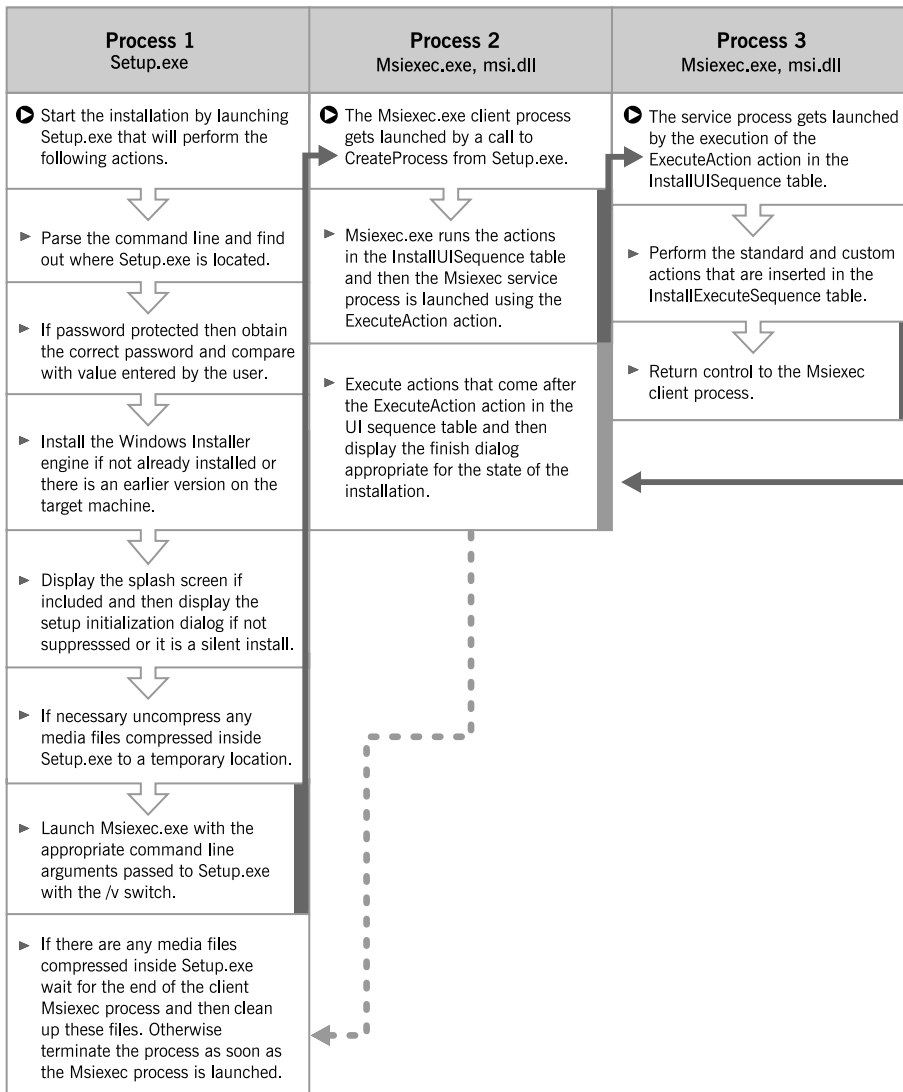


Figure 4-8: Basic MSI project run-time architecture on Windows NT/2000/XP for a fresh install with no InstallScript custom actions.

Launching a Basic MSI Project From the MSI File

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

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

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

Maintenance Install Run-Time Architecture

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

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

There is a generic issue for both a Standard and a Basic MSI project when the MSI database and the application files are compressed inside setup.exe. This issue arises when an application is installed where all files are compressed and then the end user tries to perform a maintenance operation that requires additional files to be copied to

the target machine. Additional files need to be copied when performing a Repair or a Modify operation that adds a new feature to those that have already been installed.

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

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

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

Maintenance Install Using a Standard Project

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

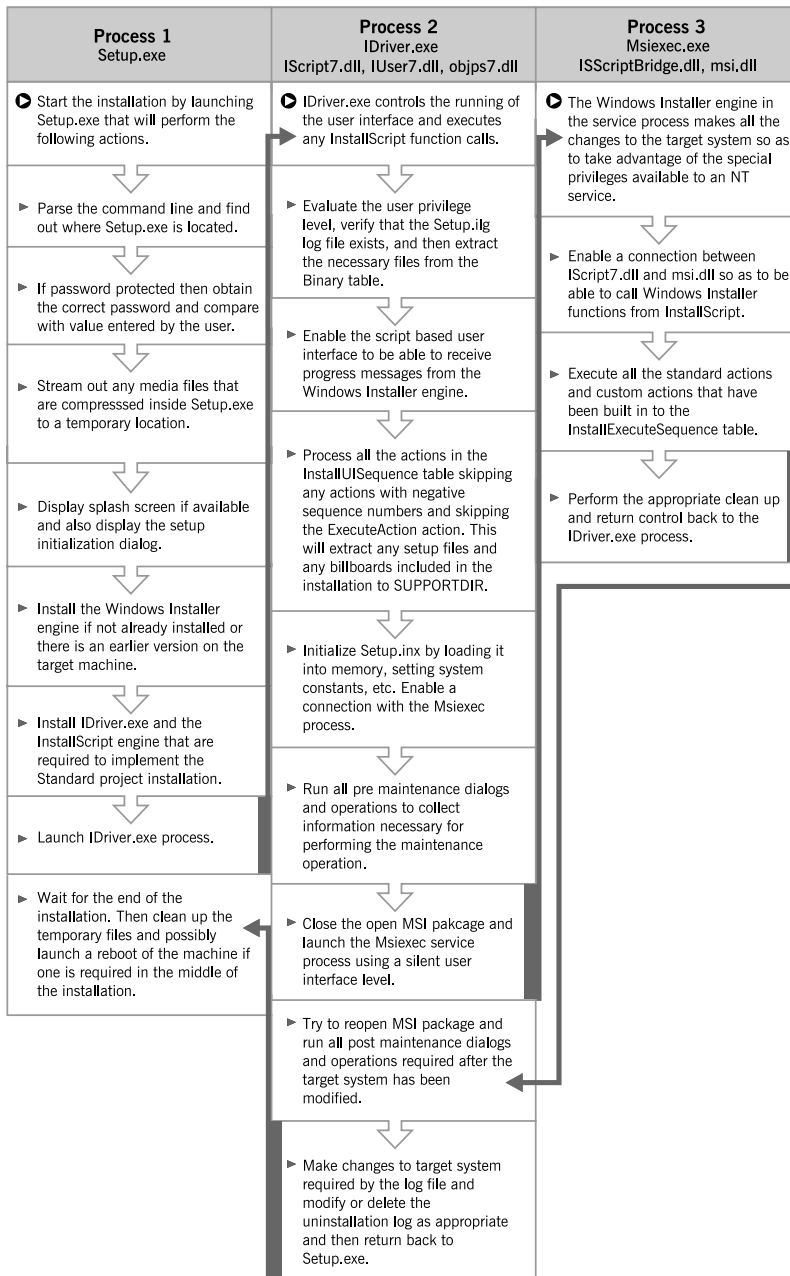


Figure 4-9: Standard project run-time architecture on Windows NT/2000/XP for a maintenance installation initiated from setup.exe.

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

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

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

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

The second difference in how the IDriver.exe process operates for a maintenance installation is instead of calling the `MsiInstallProduct` Windows Installer API function; it calls the `MsiConfigureProductEx` API function instead. During a fresh install of a Standard project, the `MsiInstallProduct` function was called from within the `program...endprogram` block. For a maintenance install, this operation is handled through a direct call by the IDriver.exe process to an event

handler function. When the DoMaintenance keyword in Setup.ini has a value of Y, the OnMaintenance event handler is called. However, when the DoMaintenance keyword has a value of N, the IDriver.exe process calls the OnUninstall event handler. These event handlers in turn call the MsiConfigureProductEx function with the appropriate command line. When the Windows Installer in the msixec.exe process has performed the requested maintenance changes to the system, control is returned to the IDriver.exe process. The event handlers are covered in more detail in Chapter 8.

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

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

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

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

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

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

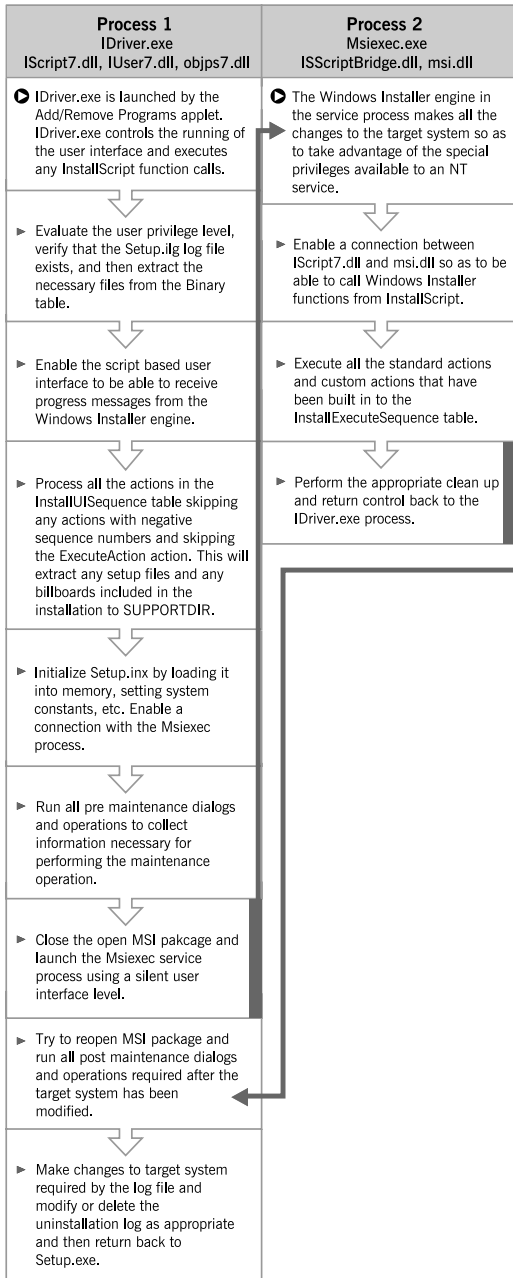


Figure 4-10: Standard project run-time architecture on Windows NT/2000/XP for a maintenance installation initiated from the Add/Remove Programs applet.

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

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

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

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

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

Maintenance Install Using a Basic MSI Project

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

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

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

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

Run-Time Architecture for Other Install Modes

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

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

Administrative Installations

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

Standard Project

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

```
setup /a
```

Basic MSI project

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

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

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

Application Advertisement

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

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

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

Standard Project

You can advertise an uncompressed Standard project on a per-machine basis by passing to setup.exe the /j switch. When you perform this operation, the run-time architecture looks very much like what is shown in Figure 4-1 for a fresh install. The main difference is that the IDriver.exe process runs all the actions that are in the AdvtUISequence table instead of the actions in the InstallUISequence table. By

default, the AdvtUISequence table has no actions inserted in it and this is the way it should remain.

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

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

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

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

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

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

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

Basic MSI Project

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

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

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

Here the optional arguments to the `/j` switch indicate whether you want to advertise the application for the current user (u) or you want to advertise the application for all

users of the machine (m). When you just use the /j switch without any arguments, you advertise for all users of the machine.

If you take care to make changes to the `OnMsiSilentInstall` event handler in a Standard project, you can advertise the .msi file created as part of the Standard project using the above command line. Just as with a Standard project, you cannot advertise a compressed Basic MSI project without first performing an Administrative install.

Localized Installations

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

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

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

```
EnableLangDlg=Y
```

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

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

Standard Project

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

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

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

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

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

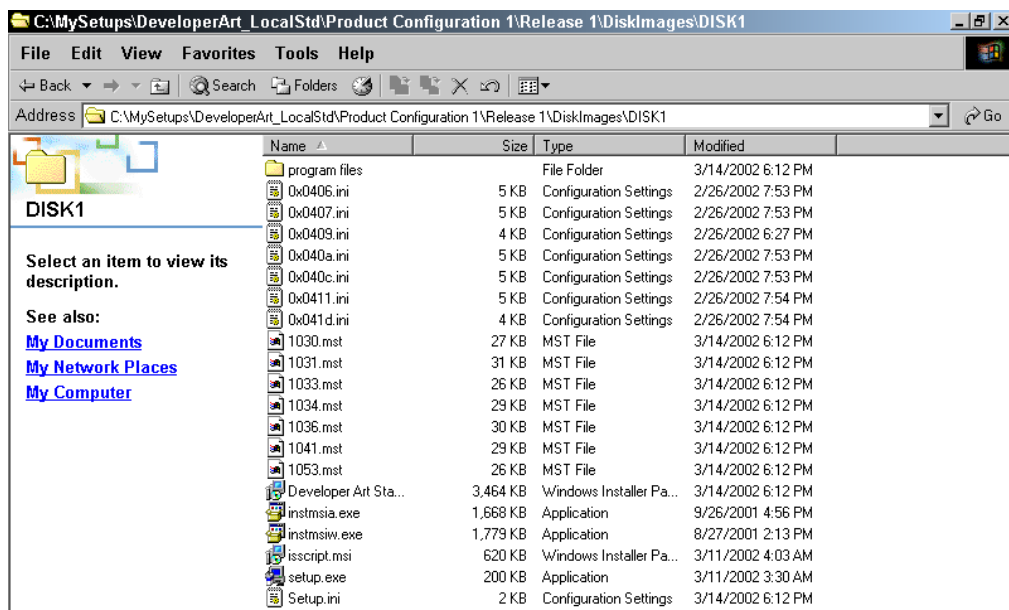
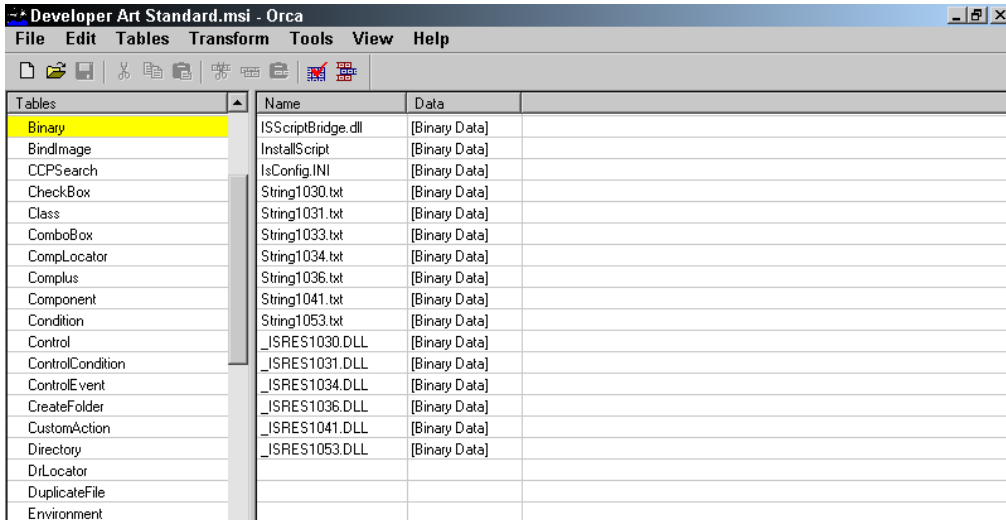


Figure 4-11: The Disk1 folder for a Standard multiple language installation project.

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

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



Tables	Name	Data
Binary	ISScriptBridge.dll	[Binary Data]
BindImage	InstallScript	[Binary Data]
CCPSearch	IsConfig.INI	[Binary Data]
CheckBox	String1030.txt	[Binary Data]
Class	String1031.txt	[Binary Data]
ComboBox	String1033.txt	[Binary Data]
CompLocator	String1034.txt	[Binary Data]
Complus	String1036.txt	[Binary Data]
Component	String1041.txt	[Binary Data]
Condition	String1053.txt	[Binary Data]
Control	_ISRES1030.DLL	[Binary Data]
ControlCondition	_ISRES1031.DLL	[Binary Data]
ControlEvent	_ISRES1034.DLL	[Binary Data]
CreateFolder	_ISRES1036.DLL	[Binary Data]
CustomAction	_ISRES1041.DLL	[Binary Data]
Directory	_ISRES1053.DLL	[Binary Data]
DrLocator		
DuplicateFile		
Environment		

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

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

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

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

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

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

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

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

```
UninstallString=C:\PROGRA~1\COMMON~1\INSTAL~1\Driver\7\INTEL3~1\
  IDriver.exe /M{ECA8C838-2A61-4956-83AF-4F3346C904C0} /11031
```

The additional argument is a "/1" followed by the language ID of the language used to perform the installation. In this example, the language ID is 1031, which indicates that German was used to perform the initial installation.

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

Basic MSI Project

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

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

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

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

```
TRANSFORMS=1031.mst
```

For maintenance operations, the Windows Installer automatically applies the cached transform so the language used is the same as what was used for the initial installation. If an end user accesses maintenance mode by running `setup.exe` again, a

language selection dialog is presented, but the selection here affects only the language used in the initialization dialog. Once the maintenance operation begins, the end user will see the language that was used in the initial installation. Running the maintenance mode from the Add/Remove Programs applet avoids the language selection dialog because setup.exe is not involved.

Run-Time Handling of InstallScript

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

Installing the InstallScript Engine

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

The InstallScript engine installation package is named `isscript.msi` and its installation is always run silently. The InstallScript engine consists of six files named `IDriver.exe`, `iUser7.dll`, `iscript7.dll`, `objps7.dll`, `_ISRES1033.DLL`, and `ISRT.DLL`. Six components are used to install these six files and these components have NULL component codes

so the Windows Installer does not know about these files after they have been installed. These files are installed to the following location:

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

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

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

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

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

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

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

The Program Block and Event Handlers

The program block has already been discussed when covering the implementation of a fresh install with a Standard project. It is now time to take a look at the program

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

```

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

//Default program/endprogram block
program

    Enable(DIALOGCACHE);

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

    ISWIPCRestoreBefore();

    ISWIONInitInstall();

    ISWIONCCPSearch();
    ISWIONAppSearch();

```

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

```

ISWIONFirstUIBefore();

ISWIONMoveData();

ISWIONFirstUIAfter();

ISWIONExitInstall();

ISWIPCRestoreAfter();

endprogram

```

Figure 4-13: *Continued.*

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

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

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

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

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

ISWIONFirstUIBefore: This function calls the OnFirstUIBefore event handler. This event handler runs the user interface for the installation. When you

add code to this event handler, the modified code is linked instead of the default implementation.

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

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

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

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

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

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

InstallScript Custom Actions

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

the function can be executed, and then pass the results of the function call back to the `msiexec.exe` process.

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

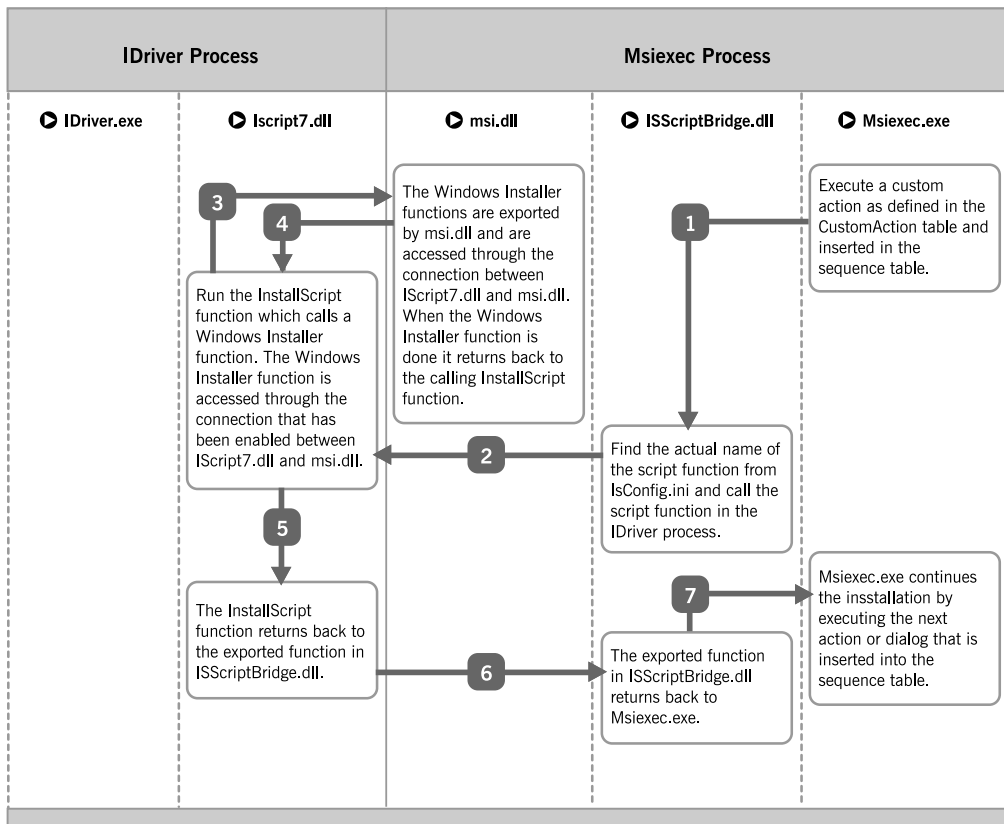


Figure 4-14: *The calling of an `InstallScript` custom action.*

For purposes of discussion, we will assume that you have an `InstallScript` custom action and the name of the `InstallScript` function that implements this custom action is `InstallNTServiceMsg`. We will also assume that this custom action makes a call to one of the Windows Installer database functions. You do all the appropriate things in order to export this function, define a custom action named `Message` where the `InstallNTServiceMsg` function is the target, and then insert this custom

action into one of the sequence tables. We are not discussing here how to create an InstallScript custom action. Chapter 11 provides a full discussion of how to create InstallScript custom actions. When you build the project, entries are made in the appropriate sequence table, in the CustomAction table, and in the Binary table.

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

The screenshot shows the Orca application window titled "Developer Art Basic.msi - Orca". The "Tables" pane on the left lists various tables, with "CustomAction" selected. The main pane displays the CustomAction table with the following data:

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

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

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

if you have more than 50 InstallScript custom actions then the version that allows 1000 custom actions is used. This is a space saving measure.

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

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

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

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

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

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

An important capability that InstallScript custom actions have that no other type of custom actions have is the ability to access the running database even from deferred

mode. In Chapter 11 we will see more about what this special capability means and how it can save you extra work.

Conclusion

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

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

Chapter

5

Creating Projects in the IDE

This chapter introduces the InstallShield Developer IDE. To help you learn the different views and functionality, you will recreate in the IDE a Standard project installation package for the Developer Art application. The creation of a Basic MSI project is left as an exercise. Completed projects for both project types can be found on the CD-ROM that is included with this book. As you work through these projects, you will see the primary differences between a Standard project and a Basic MSI project. This chapter will also acquaint you with a few of the task-oriented wizards that make setup creation easier. One of these wizards is the Release Wizard that gives you a lot of control over how you build a setup project. This chapter also looks at the Setup Best Practices Wizard and the Convert Source Paths Wizard. This book will look at the many other wizards, as you use them to perform specific tasks.

Creating a Standard Project in the IDE

Before you create a Standard project, you need to copy the source files for the Developer Art application from the CD-ROM to the following location:

```
C:\MySetups\Sources\Developer Art
```

If InstallShield Developer is not loaded you need to do that before going any further. Now make sure that the project location is set to C:\MySetups. To do this:

1. Select Options from the Tools pull-down menu.
2. Click on the File Locations tab.
3. Type C:\Setups in the Project Location field:

Setting up this location ensures that the projects on the CD-ROM will build correctly with no modification. After copying any projects from the CD-ROM to your hard drive, you need to remove the Read-only attribute from the copied files and folders. This is not necessary if you have run the installation program for the sample files.

Now do the following to create a new Standard project in the IDE without using the Project Wizard:

1. Navigate to the InstallShield Today view and click on Create a new project in the sub-view tree.
2. To create a Standard project without using the Project Wizard go to the Project Type pane and select the Standard Project icon.
3. Before clicking Create, enter the name of the project in the Project Name and Location field. The name used in the book is DeveloperArt_IDEStd to distinguish it from the project that you created using the Project Wizard.

4. Click Create to create a Standard project (Figure 5-1).

In the View List on the left side of the screen in Figure 5-1, numbers identify the steps involved in creating an installation program. Each step has one or more views under it that provide access to the location in the IDE where data required to complete the step can be entered. Following this step-by-step process allows you to build the project in a logical sequence of operations. At the bottom of the view list, after step 7, is the Advanced Views item. The views listed under Advanced Views provide most of the functionality available in steps 1 through 7. It also provides some functionality that is not available in the previous steps. This chapter discusses all of the advanced views.

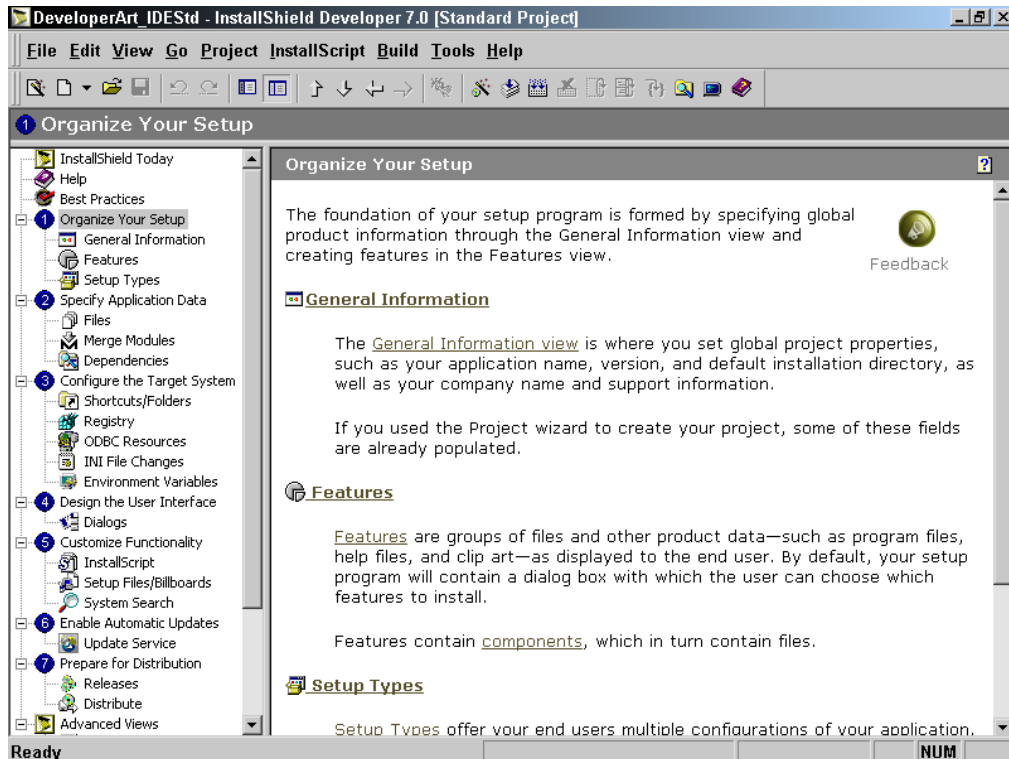


Figure 5-1: The initial view of a Standard project created in the IDE.

When you first open a new project in the IDE, the focus is placed on step 1 in the view list. This is where you begin authoring the installation program for the

Developer Art application. This project will have the same structure as the one created in Chapter 2 (Figure 5-2).

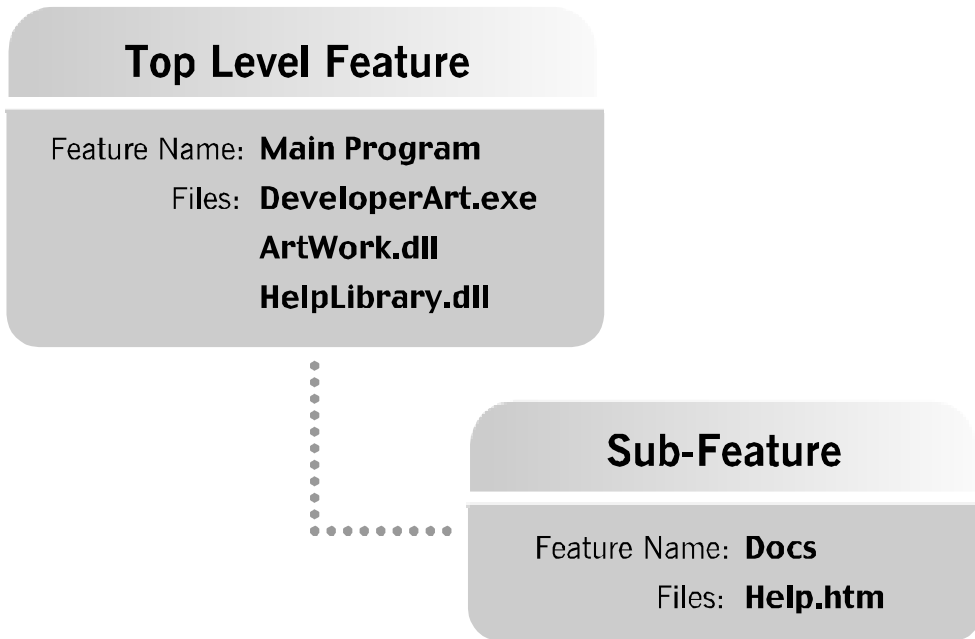


Figure 5-2: *The design of the Developer Art application.*

The Developer Art application has one top-level feature named Main Program and one sub-feature named Docs. The files installed by the Main Program feature are DeveloperArt.exe and ArtWork.dll. The files installed by the Docs sub-feature are HelpLibrary.dll and Help.htm. The file ArtWork.dll is a COM server but the other DLL HelpLibrary.dll is a standard Win32 DLL.

The creation of the installation program addresses only those areas in the View List that are required for the Developer Art application. The other areas of the View List are covered in later chapters as needed.

Organize Your Setup (Step 1)

Under Step 1, there are three required sets of operations. You need to input some general information such as the company name, application name, and setup

language. You will also define the feature tree for the Developer Art application and determine which Setup Types the installation will offer end users. Start by clicking the General Information node under the Organize Your Setup view.

The General Information View

Clicking on the General Information icon displays the General Information view (Figure 5-3).

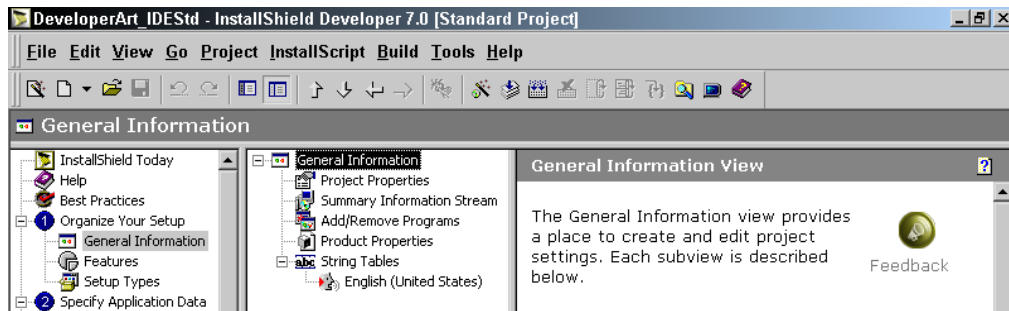


Figure 5-3: *The sub-views for the General Information view under Step 1.*

There are four sub-views where you need to enter data: Project Properties, Summary Information Stream, Add/Remove Programs, and Product Properties. We will look at the String Tables sub-view but you will not have to use it for this project.

PROJECT PROPERTIES

In this sub-view, there are four project properties that can be set or left as the default. For this project, the default values are acceptable. However, it is a good idea to get into the habit of entering values for some of these properties (Figure 5-4).

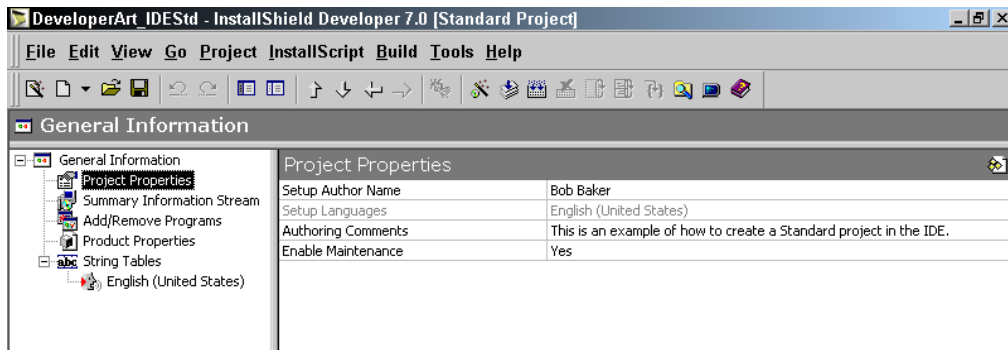


Figure 5-4: Project properties for the Developer Art installation project.

As the developer of this setup project, enter your name in the Setup Author Name field. In the Authoring Comments field, you should enter a string that allows anyone working on this project to know why it was created. The Setup Languages property provides the same functionality as the Project Wizard panel with the same name. Choosing more than one language for your project to support permits you to create installations that give the end user a choice of languages in which to run the installation.

When you click in the edit field for the Setup Languages property, a list of languages is displayed at the bottom the screen (Figure 5-5). This list of languages is more complete than the one provided in the Project Wizard. There are two check boxes that allow you to filter this list of languages so it is easier to handle. You can decide to show only the languages that have been selected for the project, or those languages that have been installed from language packs along with the languages that were added manually. Many of the languages that are shown in the complete list of languages are not available as language packs. In order to include these languages in the installation project, you need to provide them. If you select both check boxes, only the languages that have been selected for the project are displayed.

The Developer Art installation project requires only the default language, which is English. The face icon with the red arrow identifies the default language. Deselecting the default language is not permitted. If you add additional languages by selecting them in the language list, the names of these languages are listed in the Setup Languages properties field. To change the project's default language, you need to go to the String Tables view, which is covered later in this section. To remove selected

languages from the Setup Languages property field, deselect the languages in the language list.

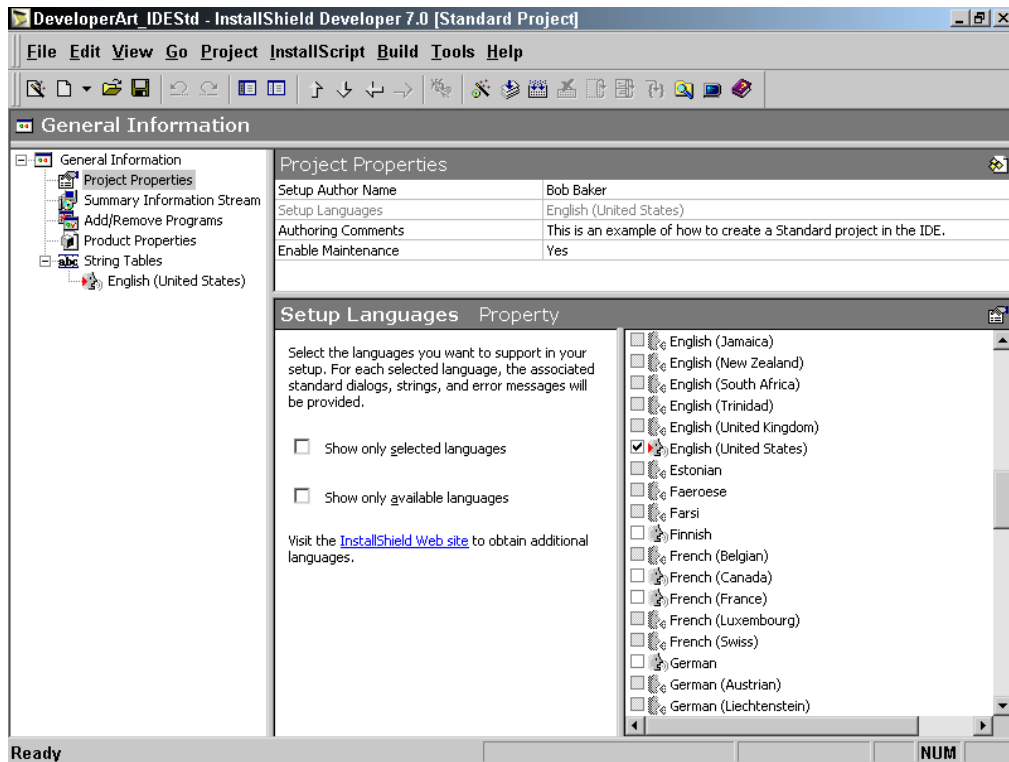


Figure 5-5: The panel for selecting the setup languages to be used in the installation project.

The Enable Maintenance property is set to Yes by default. As discussed in Chapter 2, when you run an installation after the product has already been installed, you will initiate a maintenance operation. The maintenance operation allows the end user to change the feature selection, repair the original installation, or remove the application completely. For Standard projects, setting the Enable Maintenance property to No dictates that a second running of an installation performs only a removal of the installed application. With this property set to No, a second running of a Standard installation, after the initialization, displays the Confirm Uninstall dialog (Figure 5-6). This property sets a flag in the Setup.Ini file that is created when you build a release for the application. Setup.exe uses this flag to either run a standard maintenance operation or to perform the uninstallation of the installed product.

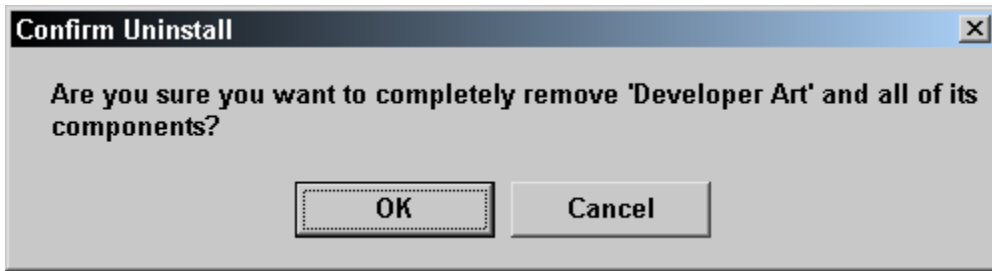


Figure 5-6: *The Confirm Uninstall dialog box when maintenance is not enabled.*

SUMMARY INFORMATION STREAM

The Summary Information Stream sub-view contains eight of the properties that make up this stream. The Summary Information Stream was discussed in detail in Chapter 3. The value for each of these properties for the Developer Art installation program is shown in Figure 5-7. Notice that the Subject and Comments properties have an associated string ID. The string ID is the identifier in curly braces ({}). These links to a String Table through a string ID allows for the localization of the Summary Information Stream. As discussed in Chapter 3, one of the properties in the Summary Information Stream is the Codepage property, which is used to translate the strings in this stream so they appear properly in Windows Explorer.

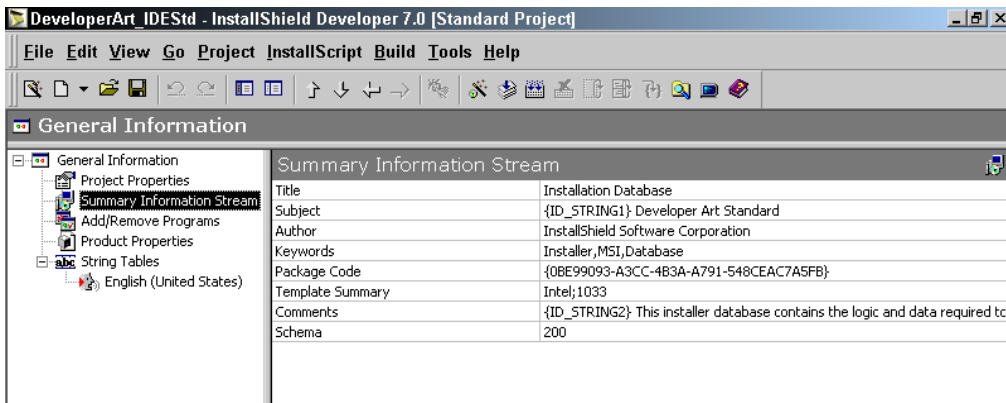


Figure 5-7: *The Summary Information Stream view for the Developer Art application.*

The Title property is a string and the one that is used here is the one that is recommended by Microsoft. This property identifies the type of file to anyone that

examines it in Windows Explorer. The Subject property provides the name of the application that is being installed. The Author property identifies the name of the company that created the installation database. The Keywords property is used to provide advanced searching for the file in Windows Explorer.

The next property in Figure 5-7 is the Package Code property, which is not the actual name of the property in the Summary Information Stream. The actual property name in the Summary Information Stream is Revision. The string "Package Code" identifies the use that is made of the Revision property. The value used for this property is a GUID. This value is used to uniquely identify the present installation package. Clicking in this property field provides the opportunity to generate a different value via the Generate GUID button near the bottom of the IDE.

The Template Summary property is used to identify the platforms and languages that the installation package supports. By default, Intel is provided as the supported platform and English as the supported language. The number 1033 is the language ID for English.

The Comments property is used to describe the general purpose of the Windows Installer database. For this project, change the default string to what is recommended by Microsoft. The complete string is as follows:

"This installer database contains the logic and data required to install Developer Art."

The final property is the Schema property. This is not the name of the property in the Summary Information Stream. The name of the property is Page Count. The name Schema indicates the use that is made of this property. The value you need to input here, in integer form, is the minimum Windows Installer version that is required to run the installation. The default value of 200 represents that version 2.0 of the Windows Installer is what is required for the Developer Art installation. Of course, since the Developer Art application is simple and does not need any of the new functionality offered by version 2.0 of the Windows Installer it could just as easily be installed with an earlier version.

ADD/REMOVE PROGRAMS

In Windows 2000 and Windows XP, there is a completely redesigned Add/Remove Programs applet, which provides a significant amount of information to the end user about applications that have been installed on their system. Prior to Windows 2000,

the only information available to the end user in this applet was one string. This string was used to identify applications on the system and was used for no other purpose, except to uninstall the identified application.

To understand where the properties will be used we need to look at the Add/Remove Program applet on Windows 2000 (Figure 5-8). The size of the application is displayed along with the frequency of use. There is a Change button and a Remove button that let the end user modify the features installed on the system for the application or directly remove the application.

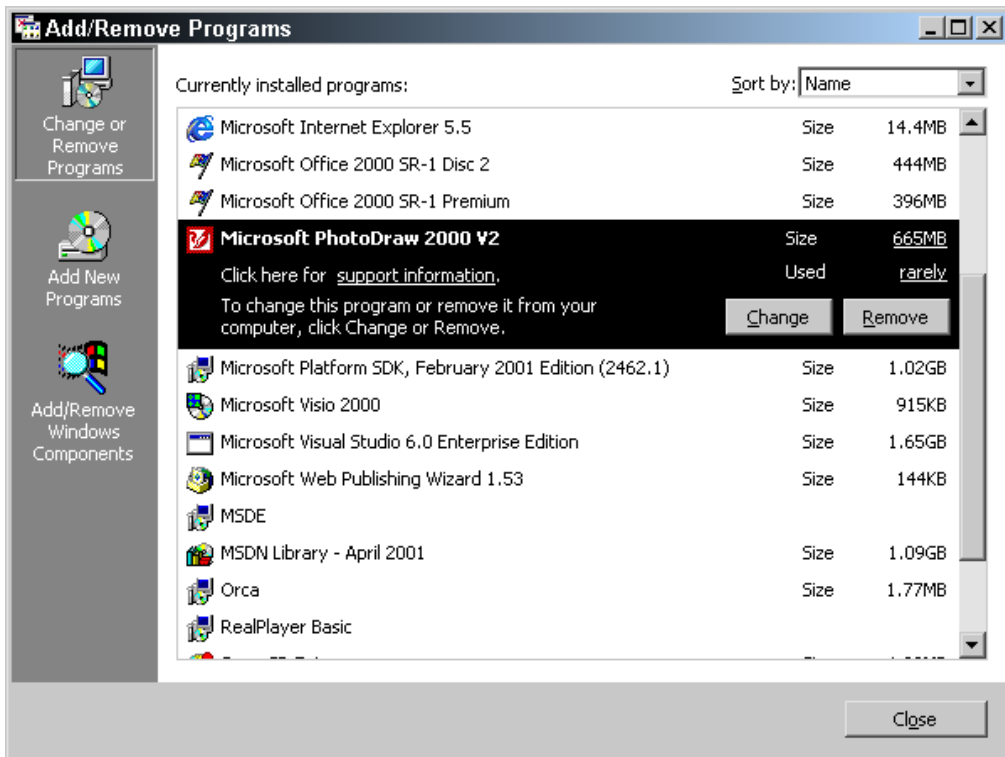


Figure 5-8: *The Add/Remove Programs applet dialog.*

Clicking on the support information link displays the Support Info dialog shown in Figure 5-9. The Support Info dialog provides information about the product, including the company that created it, the version number, a URL to the support site for the product, and a product ID. The Support Information URL is an actual link

and an end user can get to your support Web site directly from the Support Info dialog by clicking on this link.

A Repair button allows the end user to run a reinstallation to repair the application if it is not running correctly. It is the information shown in Figures 5-8 and 5-9 that is generated using the values of the properties that are entered in the Add/Remove Programs sub-view.



Figure 5-9: *The Support Info dialog of the Add/Remove Programs applet.*

There are 12 Add/Remove Programs-related properties that you can set as necessary (Figure 5-10). The first property is to identify any special icon that you want to display beside your applications name in the Add/Remove Programs applet. If you click in this field, a button to the right with an ellipsis allows you to browse for a file and then select the icon from that file that you want to use. For the Developer Art application, browse to the DeveloperArt.exe file and select the icon with the index 0. To see a bigger selection of icons, you could browse to the SHELL32.DLL file under the %SystemRoot%\System32 folder. On Windows 2000, this file offers 106 different icons.

The next three properties allow you to disable the three buttons that appear in either the Add/Remove programs applet dialog or the Support Info dialog. Note that when the Change and/or Remove buttons are disabled, they are still visible in the dialog, but they are not functional. Setting the Disable Repair Button property hides this button on the dialog. The remaining eight properties are described by the property names.

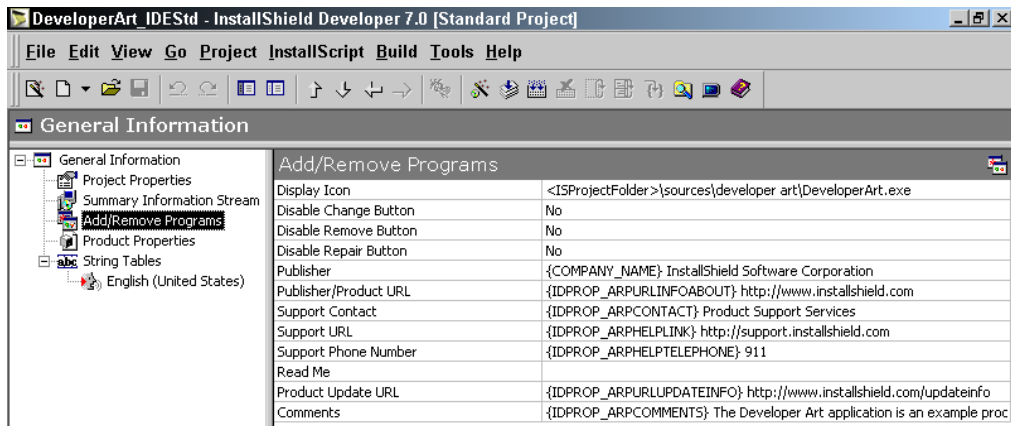


Figure 5-10: *The properties for the Add/Remove Programs sub-view.*

Notice the identifiers inside the curly braces. These are string IDs that tie these strings into the string tables. For the Read Me property, the property value is NULL because there is no readme file for the Developer Art application. For the Comments property, type an appropriate string such as "The Developer Art application is an example product for learning about the use of InstallShield Developer." For the remainder of the properties, you can leave the default values.

PRODUCT PROPERTIES

The most important properties in the General Information view are those in the Product Properties sub-view (Figure 5-11).

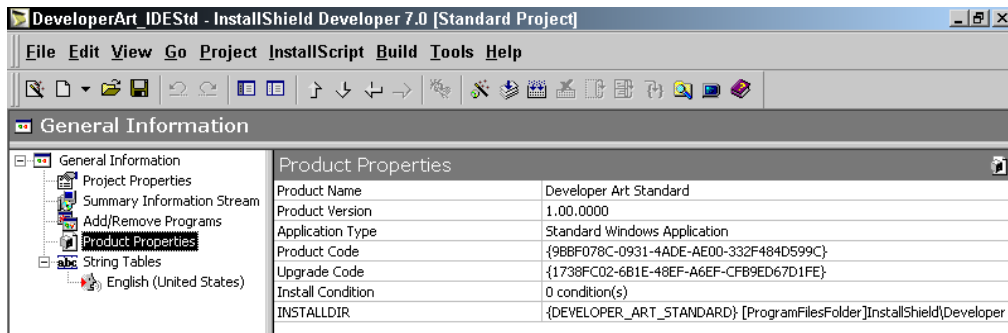


Figure 5-11: *The properties in the Product Properties sub-view.*

The Product Name property is used to set the ProductName property in the Property table and it is also used to name the MSI file that is created when the project is built. The ProductName property is one of the required properties in the MSI database. For this project, use the string "Developer Art Standard" as the value for this property. Leave the default value for the Product Version property. The value here is used to set the ProductVersion property in the Property table and it is also one of the required properties.

The Application Type property is not used in the MSI database. This information resides in the project file for your information only. Clicking in the field for this property displays an editable combo box with four choices: Standard Windows Application, Internet Application, Database Application, and Client-Server Application. For the Developer Art application, select Standard Windows Application.

The next two properties, the Product Code and Upgrade Code, are GUIDs. Default values are generated when the project is created. Depending on the project, you might want to change these codes. One way to get a new GUID is to click in the field and then click on the Generate GUID button near the bottom of the IDE. Clicking this button generates another unique GUID created at the time the button is clicked. However, sometimes you might want to use a specific GUID and then you have to copy and paste it from another location, such as another project file.

The Product Code property is used to set the ProductCode property in the Property table of the database. This is one of the properties in the Property table that must be set. The ProductCode property is the principal identification of an application or product. There are many rules about when the ProductCode property has to be

changed and when it can be left the same. Changing the ProductCode property becomes an issue when performing upgrades of a product.

The Upgrade Code property is used to set the UpgradeCode property in the Property table. This is not a required property, but it is recommended that this property be set; otherwise, you cannot perform major upgrades using the mechanisms provided by the Windows Installer. The UpgradeCode property is used to represent a related set of products. This property is used in the Upgrade table to search for related versions of the product that are already installed on the target system.

The Install Condition property allows you to define any checks that need to be made of the installation environment. Click in the field for this property to display a small ellipsis button at the right of the field. Clicking this button displays the Product Condition Builder dialog (Figure 5-12). In this dialog, you can add any conditions that you want checked when the installation starts.

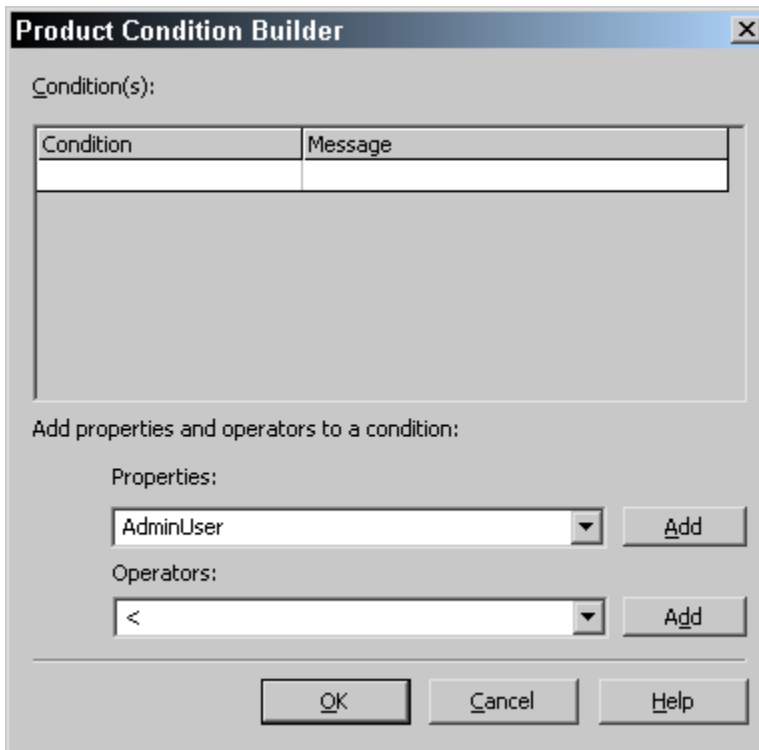


Figure 5-12: *The Product Condition Builder dialog box.*

A condition as entered here consists of the condition itself and a message that will be displayed by the Windows Installer if the condition does not evaluate to TRUE. When you create conditions here, you are making entries that will be rows in the LaunchCondition table. Part of this dialog contains a condition builder capability that allows you to choose properties and operators from a drop-down combo box to build a condition. You can also type directly in the condition field without using the condition builder.

The final property that you need to define is the default for the root install location for the application. The name of this property is INSTALLDIR. As discussed in Chapter 3, INSTALLDIR is an identifier in the Directory table that is made into a path property during file costing. By default, all features and components have INSTALLDIR as the destination. For this project, set the company name to InstallShield and the application name to Developer Art Standard (Figure 5-11).

STRING TABLES

String tables enable the easy localization of the installation program's user interface. For the Developer Art application, only English is selected as the setup language so only an English (United States) string table appears. Click on the English (United States) node to view the associated string table (Figure 5-13).

A string table has four columns. The first column is the identifier by which a particular string will always be associated and the second column is the string itself. The third column allows you to make comments if there is something special about the string itself. The fourth column is the date when the string was created or was last changed. In the string table shown in Figure 5-13, the entries in the string table are sorted with the string created last displayed at the top. You can sort the string table by clicking on the header for any of the columns. Sorting by the Modified column is a good way to separate the default strings and the custom strings into different groups.

As shown in Figure 5-13, only two custom strings have been entered at this point. These custom string IDs have the generic format ID_STRINGX where X is a sequence number that is incremented for each new string we add to the project. You can export string tables in total or by selected rows. You can also import string tables.

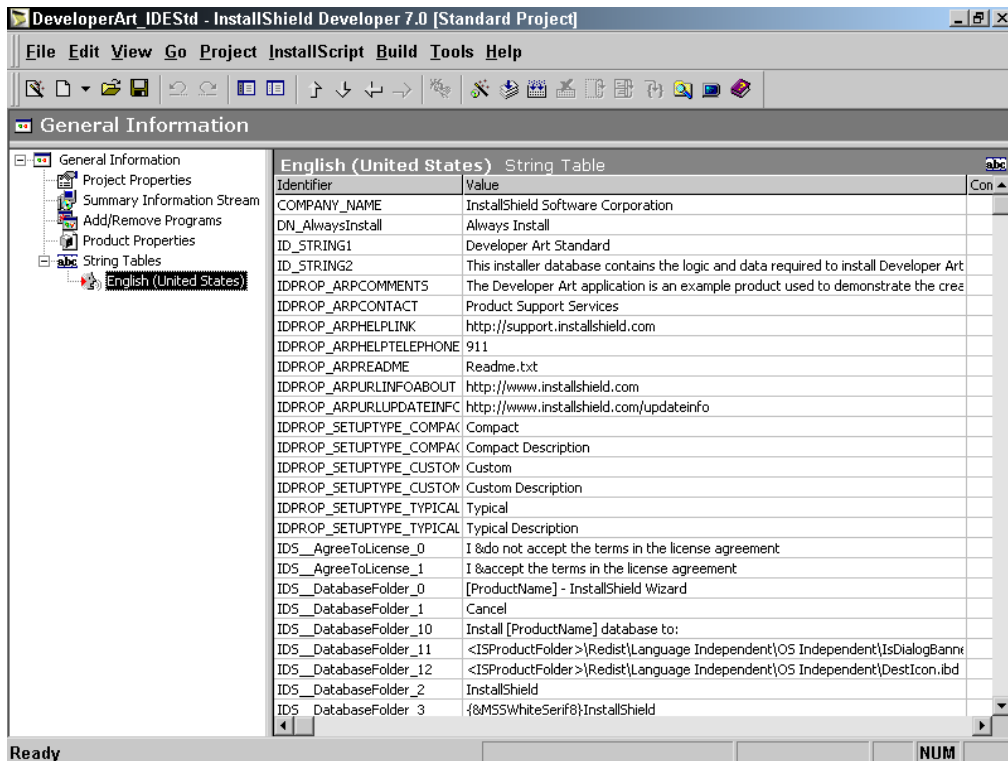


Figure 5-13: *The English (United States) string table for the Developer Art application.*

You have now entered the properties that make up the General Information sub-view of the project. The second part of Step 1 is to define the features that describe the logical structure of the application.

The Features View

There are only two features in the application, one top-level feature and one sub-feature under that (Figure 5-2). Figure 5-14 shows a view of the default feature that is part of any project created directly in the IDE. The name of this default feature is DefaultFeature.

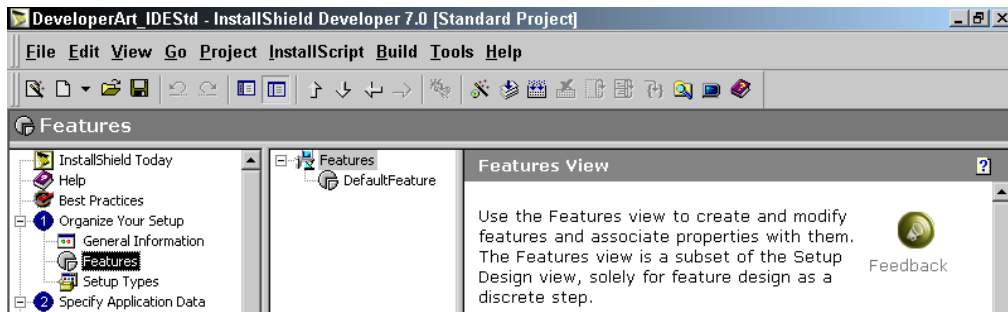


Figure 5-14: *The Features view and default feature tree under Step 1.*

The first thing that you want to do is rename the default feature to MainProgram and then add the sub-feature to this feature. You can rename the default feature by right-clicking and selecting Rename from the context menu.

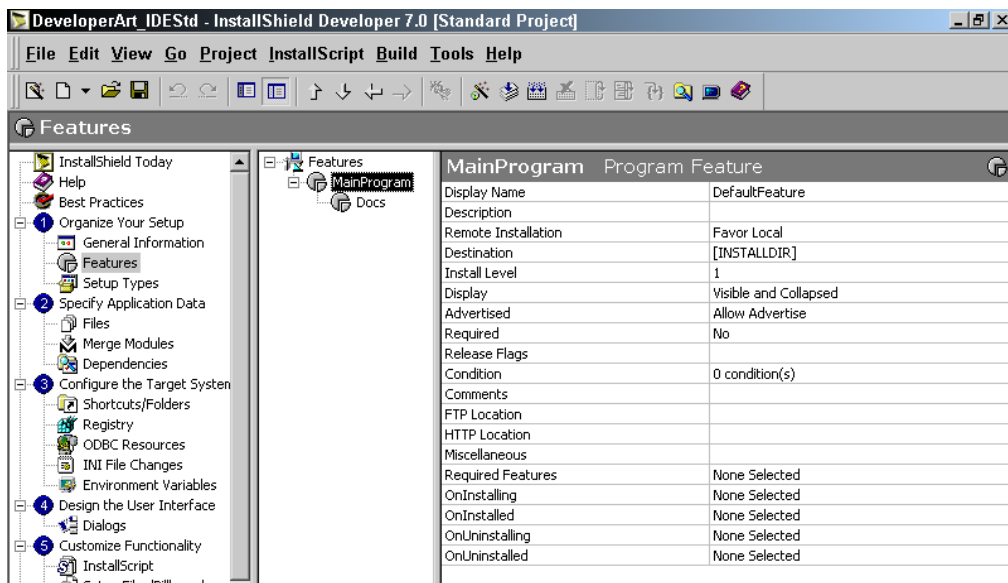


Figure 5-15: *The Features view showing the feature tree for the Developer Art application.*

You can add a sub-feature by right-clicking on the top-level feature and selecting the New option. Name this sub-feature Docs. Right-click on a feature to display the context menu. The options allow you to configure the structure of the feature tree, in addition to adding, renaming, and deleting features. After you have created the feature

tree for the Developer Art application, you should see something like what is shown in Figure 5-15.

To the right of the feature tree, there is a panel where you can set certain attributes for the feature that is highlighted in the feature tree. These attributes are important in the creation of a robust installation program. Many of these attributes are used to populate the Feature table when you build the project.

The rest of the feature attributes are used only in Standard projects and define certain aspects of the programming approach used with these project types. After we discuss these attributes, you will make a few changes in them for each of the features.

Display Name: The display name defines the string that is used in the custom setup dialog in the installation user interface to show the name of the feature. The default value for this property is the name of the default feature, which is DefaultFeature. This property is not changed if you rename the feature in the feature tree. The Display Name property is used to populate the Title column of the Feature table. This column is allowed to be NULL and if it is NULL, then no feature name is displayed in the custom setup dialog.

Description: The value of this property is used to display a description of the selected feature in the custom setup dialog. When the end user highlights a feature in the custom setup dialog, the string entered here explains the feature's purpose. The value here is used to populate the Description column of the Feature table. This value is allowed to be NULL.

Remote Installation: As discussed in Chapter 3, the Feature table contains an Attributes column that is used to set the feature's default state. The Remote Installation property is used to set part of the attribute value for the feature. The Attributes column for a feature is made up of a number of applicable bit-flags. This property is just one aspect of a feature's state that is entered into the Attributes column. The values for this property are Favor Local, Favor Source, and Favor Parent. The Favor Local option means that the default action will be to install the components of the feature to the local hard drive. The Favor Source option means that the components of the feature will be installed to run from the source media. In other words, these components will not be copied to the local hard drive. The Favor Parent option is applicable only for sub-features and it means that, by default, the sub-feature will be installed using the same state as the parent feature. The Remote Installation property is also available for components.

In fact, what the component does and what the feature's default state is depend on the settings for the components and this setting for the feature.

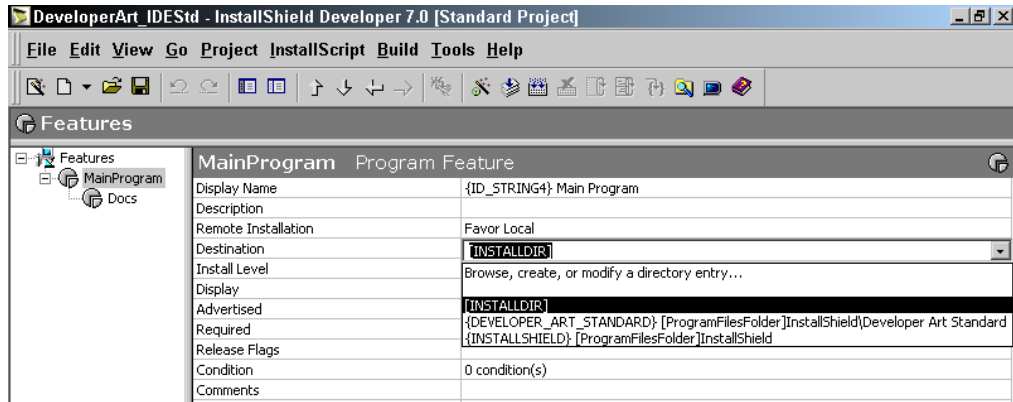


Figure 5-16: *The options for specifying a feature's Destination property.*

Destination: This is the default destination for the feature and it is a foreign key into the Directory table. The default value for this is [INSTALLDIR], but it can be any key into the Directory table. However, it should be a key to a Directory table identifier that does not have a fixed location on the target system. Otherwise, the end user will not be able to browse to a different location for installing the feature.

If you click in the Destination field, a drop-down combo box displays the five possible selections (Figure 5-16). The default destination for the product, [INSTALLDIR], was set under the Product Properties section of the General Information view. The Choose Folder dialog in the installation user interface points to this location. The location for this project is:

```
{DEVELOPER_ART_STANDARD} [ProgramFilesFolder] InstallShield\  
Developer Art Standard
```

Inside the curly braces, there is an identifier that points to a row in the Directory table. Up to this point we have seen that string IDs are held inside curly braces. This is a new use of the curly brace. Consistency might indicate that the INSTALLDIR identifier should be shown inside curly braces, but the use of the square brackets is a holdover from earlier InstallShield products to be consistent with its previous use. The DEVELOPER_ART_STANDARD identifier points

to the same location that `INSTALLDIR` points to. The `INSTALLSHIELD` identifier points at a location that is one level higher than that which is pointed to by the `INSTALLDIR` or by the `DEVELOPER_ART_STANDARD` identifiers.

There is also what looks like a blank line in the drop-down combo (Figure 5-16) and this allows you to set the destination for a feature to `NULL`. This is allowed because the `Directory_` column in the Feature table can be `NULL`. Select the “Browse, create, or modify a directory entry” option from the drop-down combo box to launch the Browse for Directory dialog (Figure 5-17).

This dialog was first discussed in Chapter 2 when you launched it from the Application Features panel of the Project Wizard. This dialog allows you to create new identifiers for different locations.

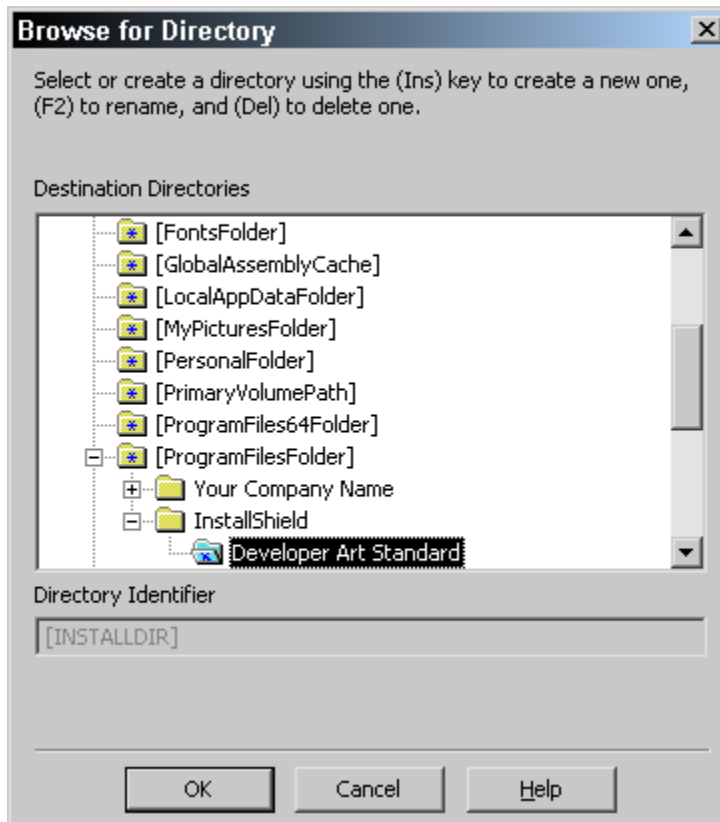


Figure 5-17: *The Browse for Directory dialog to define a new feature destination.*

These identifiers, just like `INSTALLDIR`, are converted into properties during file costing and the resolution of the Directory table. This was discussed in Chapter 3. For your introduction to creating projects directly in the IDE, use the default value of `[INSTALLDIR]`. There is an example in Chapter 14 of how to create your own directory identifier for a feature.

Install Level: The value of this property is used to populate the Level column of the Feature table. The use of the Level value in the Feature table was discussed in Chapter 3 as the means by which setup types can be defined. This number can be any integer from 1 to 32,767. This number is compared to the value of the `INSTALLLEVEL` property in the Property table. The `INSTALLLEVEL` property defines the initial install level for the package as a whole. If the Level value for the feature is equal to or less than the `INSTALLLEVEL` value, the feature is installed by default; otherwise, the feature will not be installed. If this value were set to zero, the feature would be hidden from the end user and not installed. For a Standard project, this value is not used unless there are no defined setup types.

Display: This property allows you to set how the initial display of the feature tree appears in the custom setup type dialog in the installation user interface. Click in the property's field to access a drop-down combo box that presents three options: Not Visible, Visible and Expanded, and Visible and Collapsed. Not Visible indicates that the feature will be hidden and the end user cannot interact with it in the custom setup dialog. A hidden feature will be installed or not installed based on how its Level value compares with the value of the `INSTALLLEVEL` property. The Visible and Expanded option displays the expanded feature tree in the custom setup dialog. This is pertinent only when the feature has sub-features. The default option, Visible and Collapsed, means that the feature tree is initially shown collapsed. In order to see the sub-features, the end user needs to expand the tree.

Advertised: The value of this property goes into setting the Attributes column of the Feature table, just as was done with the Remote Installation property discussed above. Advertising a feature in a Basic MSI project allows that feature to be installed on demand. This property is not used for a Standard project.

Required: This property defines whether a feature is required or not. A required feature cannot be deselected. To set the value, click in the field and select Yes or

No from the drop-down menu. This is another of those properties that helps to set the final value for the Attributes column in the Feature table.

Release Flags: Up to this point, every feature property relates to how a feature appears in the custom setup type dialog of the installation user interface. The Release Flags property, however, has nothing to do with anything that goes into the Feature table. A release flag is a build-time mechanism that allows you to create different builds from the same project. You can use release flags to filter out different features for different builds. This can come in handy if you want to generate an evaluation build, a light version build, a full version build, etc. for your application.

Condition: The Condition property allows you to define a condition for a feature. If the condition evaluates to TRUE, the Level value in the Feature table will be changed to a new value as specified as part of the condition.

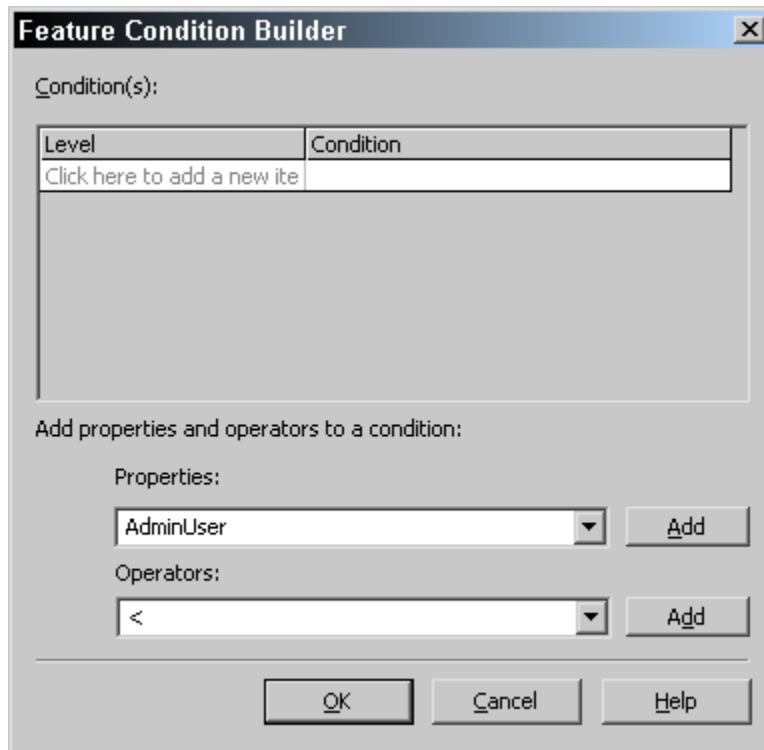


Figure 5-18: *The Feature Condition Builder dialog.*

Click in the field for this property to display an ellipsis. Click on this button to display the Feature Condition Builder dialog (Figure 5-18).

This dialog is similar to the dialog that is displayed to create an install condition as part of the Product Properties under the General Information view. The difference here is that you can specify a level to which the Level value property for the feature will be set if the condition evaluates to TRUE. Any conditions that you define here are entered into the Condition table of the database. These conditions are evaluated during the file costing operation.

Comments: This property is used to provide comments in the project file that are important to understanding the feature. These comments are saved only in the project file, and do not affect the installation program created from the project file.

FTP Location & HTTP Location: These two properties were created as part of the InstallShield Professional 5.0 functionality and the original purpose of these properties was to allow for the update of features using a download-on-demand capability via the Web. This is no longer the planned use of these properties since newer and better capabilities have been developed for dynamic updating. Presently these properties are used by setup developers to store strings that provide information that can be accessed at run time by calling a certain function in InstallScript. For a Standard project, these have to continue to be supported because of the migration of installation programs created by InstallShield Professional products. When creating projects in InstallShield Developer and not upgrading from an earlier project, you should not use these properties.

Miscellaneous: This property is where you place any string data that you want to retrieve at run time. You can retrieve this data using the FeatureGetData InstallScript function. This property should be used to store data that you want to retrieve at run time in place of using either the FTP Location or the HTTP Location properties for this purpose.

Required Features: This property is to allow you to tie features together. If the end user selects a particular feature, the installation will automatically install any other features that are necessary for the selected feature to function properly. This is something that is very difficult to do in a Basic MSI project. Click in the

Required Features field to display the ellipsis button. Click on this button to display the Required Features dialog box (Figure 5-19).

The Required Features dialog displays a tree of all of the features defined in the project. The feature from which this dialog was launched is selected and all the other features are deselected. You can select the other features that need to be installed when the selected feature is installed. There is no mechanism in the IDE that you can use to create an exclusive relationship between two features, where a feature is deselected when the end user selects another feature.

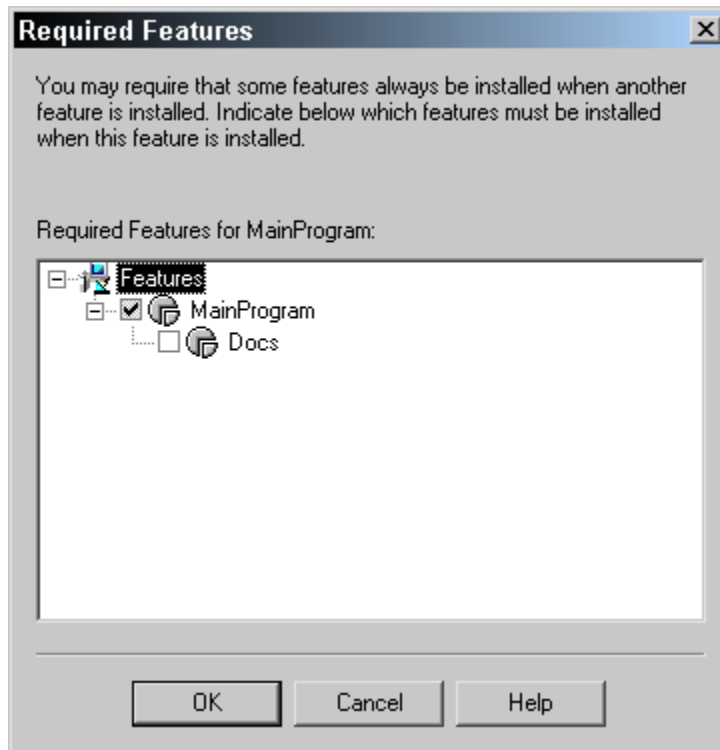


Figure 5-19: *The Required Features dialog box.*

OnInstalling: This property identifies a function that is to be called before installing the feature. This type of function has to be prototyped in a special fashion. When you click in the field, a drop-down combo box provides a list of all the functions that have the correct prototype. You can select the function that

you want called. For functions to be available in the combo box they need to first have been declared in your script.

OnInstalled: This property identifies a function that is to be called after installing the feature. This type of function has to be prototyped in a special fashion. When you click in the field, a drop-down combo box provides a list of all the functions that have the correct prototype. You can select the function that you want called. For functions to be available in the combo box they need to first have been declared in your script.

OnUninstalling: This property identifies a function that is to be called before uninstalling the feature. This type of function has to be prototyped in a special fashion. When you click in the field, a drop-down combo box provides a list of all the functions that have the correct prototype. You can select the function that you want called. For functions to be available in the combo box they need to first have been declared in your script.

OnUninstalled: This property identifies a function that is to be called after uninstalling the feature. This type of function has to be prototyped in a special fashion. When you click in the field, a drop-down combo box provides a list of all the functions that have the correct prototype. You can select the function that you want called. For functions to be available in the combo box they need to first have been declared in your script.

Now that you have learned about the properties that can be set for a feature in a Standard project, you can set a few of these properties for the Developer Art application. You need to change only two of the properties for each of the features in the project: the Display Name and the Description properties. The values for each of these properties are as follows:

MainProgram Feature

Display Name: Main Program

Description: This feature installs the main executable for the Developer Art application.

Docs Feature

Display Name: Docs

Description: This feature installs the help files for the Developer Art application.

For the time being, you do not have to change any of the other properties. However, you should take the opportunity to come back to these properties later and experiment with them.

The Setup Types View

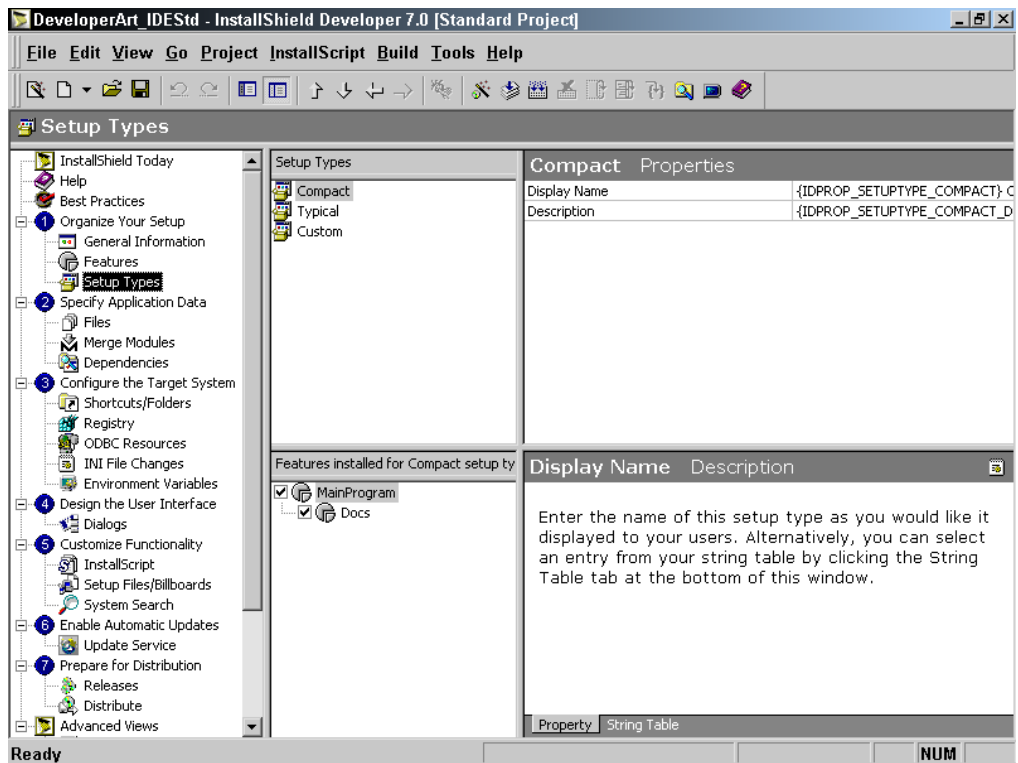


Figure 5-20: *The Setup Types view for the Developer Art application.*

The last view under Step 1 is the Setup Types view where you can indicate the setup types that you want your application to offer the end user. The Setup Types view

provides three default setup types (Figure 5-20). A setup type provides for the installation of specified features when the end user selects the setup type in the installation user interface. The dialog that presents the setup types to the end user is usually called the SetupType dialog.

The Setup Types view displays a screen that is divided into four panels. In the upper-left panel is a list of three default setup types that were generated when the project was created. Right-click in this panel to display a context menu that provides options to Add, Rename, Remove, Move Up, and Move Down. Using this functionality you can create your own setup types and rearrange them.

The upper-right panel contains the property sheet where you can create or modify the properties for each of the setup types. There are only two properties for any setup type, Display Name and Description. Each of these properties is tied to a string ID in the string table. This allows you to easily localize the installation user interface.

In the lower-left panel, you can set the features to be installed for each of the setup types that have been defined in the Setup Types panel. For each of the three default setup types for the Developer Art application both features are selected by default. You can leave this default selection for this project. The lower-right panel provides help for the two properties that are associated with each of the setup types.

This completes our discussion of the three views that compose Step 1 of project creation. For the remainder of the steps, we need to discuss a few of the associated views. Most of the views are not required for the Developer Art application because it is a simple project.

Specify Application Data (Step 2)

Three views compose this step in the project creation process. For the Developer Art project, however, you need to use only the Files view. The other views, Merge Modules and Dependencies, are discussed in Chapters 13 and 14. When we discuss the Files view, you will also learn about the Setup Best Practices Wizard. To be able to do this, you need to make sure that you are enforcing best practices when you create components. Do this by selecting Options from the Tools drop-down menu. This launches the Options dialog. On the General tab of this dialog, ensure that the Enforce Setup Best Practices check box is selected.

The Files View

Next, click on the Files icon in the View List to display the Files view (Figure 5-21). As with the Setup Types view, the Files view is divided into four panels. Above these four panels is a drop-down menu that lists all the features in the project. The upper-left panel provides a view of the file system on the build machine, the machine on which you are creating the project. You can browse through the file system in this panel just as in Windows Explorer. The upper-right panel contains a list of all the files in the folder that is selected in the upper-left panel.

The lower-left panel is where you describe destinations on the target computer where you want to copy files. In this panel, you can define the tree of folders that will be affected by the installation. By default, the folder tree is displayed as defined by the default value of the INSTALLDIR property.

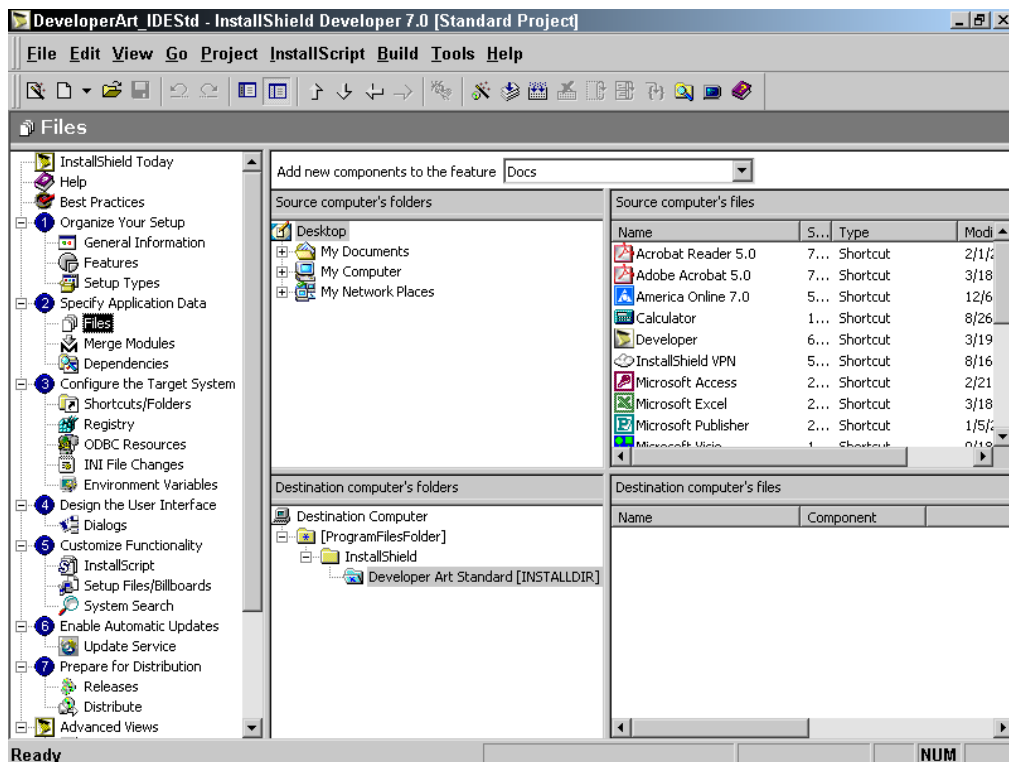


Figure 5-21: The Files view for the Developer Art application.

The lower-right panel provides a list of the files that are going to be copied as part of the installation. In Figure 5-21, there are no files because none have been designated yet.

For the Developer Art application, you need to copy all files to the location pointed at by `INSTALLDIR`. Before you drag and drop any files, select the feature that will install these files from the drop-down combo box at the top of the view.

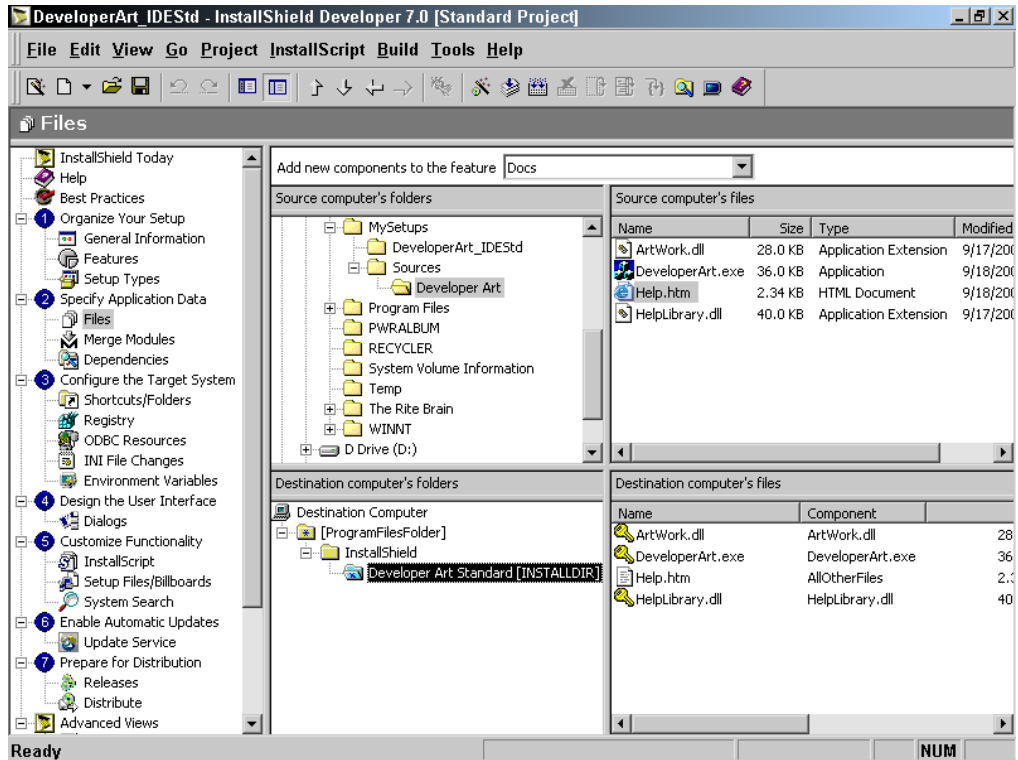


Figure 5-22: *The Developer Art application files as seen in the Files view.*

To copy the Developer Art application files:

1. Select the MainProgram feature from the feature drop-down menu.
2. In the upper-left panel, browse to the source file location for the Developer Art application.

3. From the upper right panel, drag and drop the DeveloperArt.exe, ArtWork.dll, and HelpLibrary.dll files into the INSTALLDIR location in the lower-left panel.
4. Select the Docs feature from the feature drop-down menu.
5. From the upper right panel, drag and drop the Help.htm file into the INSTALLDIR location in the lower-left panel.

When you finish copying the Developer Art application files to the INSTALLDIR location, the Files view should look similar to what is shown in Figure 5-22.

There is a lot of hidden functionality in the lower-left panel of the Files view. If you right-click on the Destination Computer string the context menu appears as seen in Figure 5-23. There are a number of options available from this context menu, many of which are disabled at this time.

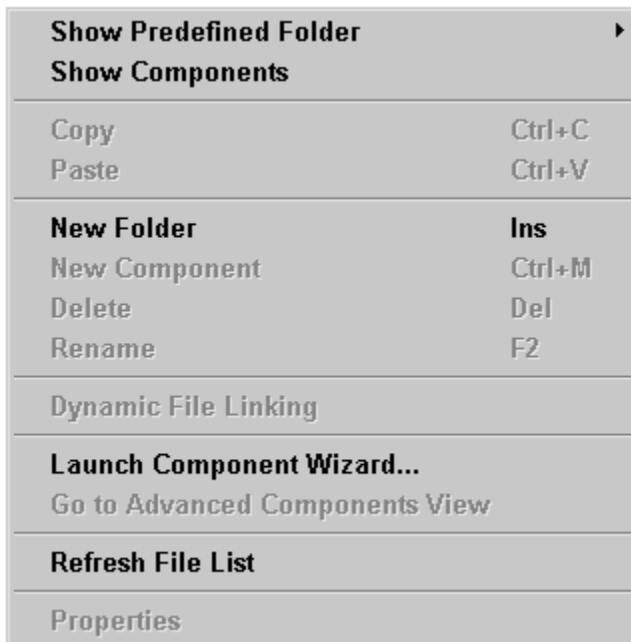


Figure 5-23: *The files location context menu.*

By default, only the locations on the target computer are displayed in the lower-left panel under the Destination Computer. To see the components that were created when you dragged and dropped the files into the INSTALLDIR location, right-click on Destination Computer and select Show Components. Expand the tree under INSTALLDIR to see a list of components in the lower-left panel.



Select the Show Predefined Folder option to see a sub-menu, shown on the next page, which lists all the predefined folders that are available from the operating system. Note that the [ProgramFilesFolder] option is checked and disabled on this sub-menu. This is because this folder is already in use as part of the definition of INSTALLDIR. You can use this functionality to create destinations for new components. You can create a new destination and drag and drop files into this location. New components will be created. If you choose one of these predefined folders, it will appear under the Destination Computer location in the lower-left panel. When you do this, a default component is created, AllOtherFilesX where the X stands for some sequence number.

The New Folder option allows you to add folders, under the predefined folders or directly under the Destination Computer location. For every folder added, another default component is created. In most cases, you will delete these components since you probably will not be using them. For folders that you name, you can directly delete them or delete the components that are under them. To delete a predefined folder, delete the components and folders that are under it. You cannot delete the entire tree

under a predefined folder directly. You can, however, delete an entire tree of folders that you have defined by right clicking and selecting Delete or press the Delete key on the keyboard.

One of things that you need to do for the Developer Art project is delete any unnecessary components that have been created. Ensure that the Show Components option is active and expand the list of components under the INSTALLDIR folder (Figure 5-24). In the lower-left panel, select the INSTALLDIR folder to see a list of all the files that are going to be installed with the Developer Art installation in the lower-right panel.

In the lower-left panel, select each component in order to verify what files will be installed by that component alone. This is done to find any components that do not contain any files. The component that does not contain files is named DefaultComponent. Because you do not want any unnecessary components in your project, right-click on this component and select Delete.

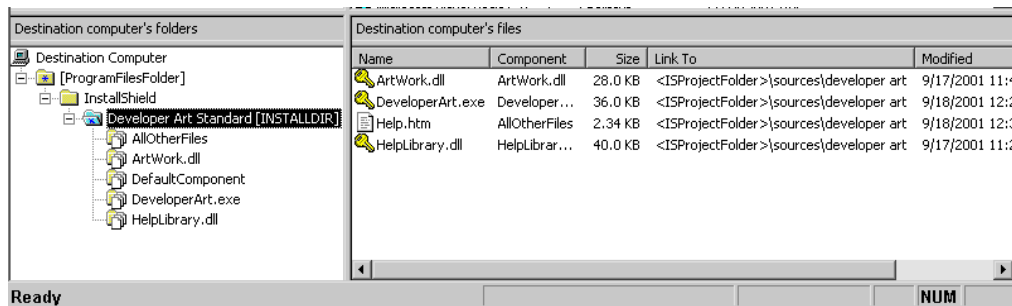


Figure 5-24: *The file list for the Developer Art installation.*

In the lower right panel right-click on a file to display the context menu. These options were discussed in Chapter 2 when we covered the Application Files panel of the Project Wizard. For the Developer Art project, right-click on the Help.htm file and select Set Key File to set it as the key path of its component, the AllOtherFiles component. A yellow key icon appears next to this file to indicate that it is the key path for the component that will be installing it

We now want to get acquainted with the Setup Best Practices wizard before we move on to Step 3.

The Setup Best Practices Wizard

At the beginning of this chapter, you ensured that the Enforce Setup Best Practices option was selected on the Options dialog. Because this option is set, the Setup Best

Practices wizard will appear if one of the rules for creating components is broken while you create your project.

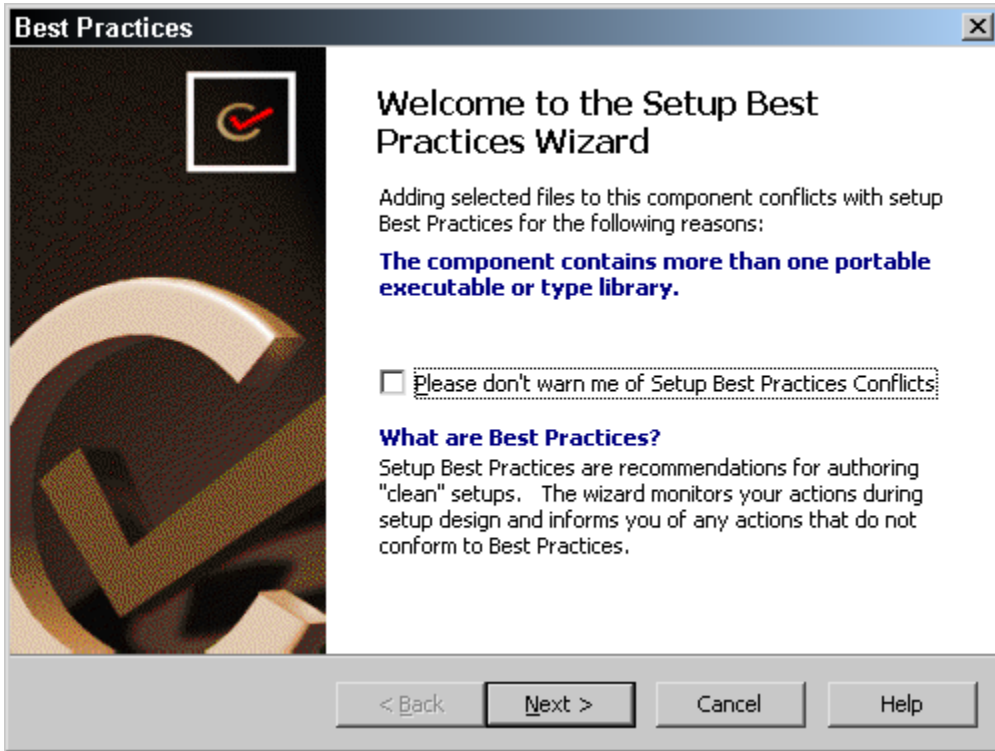


Figure 5-25: *The Welcome dialog for the Setup Best Practices wizard.*

These rules were discussed in Chapter 3 and it is the first three of these rules that are addressed by the Setup Best Practices wizard. These three rules are:

1. Every .exe, .dll, and .ocx file needs to be in its own component and this file has to be designated as the key path for the component.
2. Every .hlp and .chm file needs to be in its own component. These files need to be designated as the key path for the component. The associated .cnt and .chi files need to be added to the component that is installing their associated .hlp or .chm file.

3. Never create a component for a file and other resource that is already available in a merge module. A brief description of merge modules is provided in Chapter 14.

The way to launch the Setup Best Practices wizard is to break one of the above rules. You can do this very easily from the Files view by dragging and dropping the HelpLibrary.dll file into the component that contains the ArtWork.dll file. Since we are not supposed to have two DLLs in the same component, the Setup Best Practices wizard appears (Figure 5-25).

In the Welcome panel, you can select the “Please don’t warn me of Setup Best Practices Conflicts” check box and click Next to move to the next panel in the wizard. Or you can just click Next to go to the next panel. The difference in the two actions is that the first action deselects the Enforce Setup Best Practices option on the Options dialog.

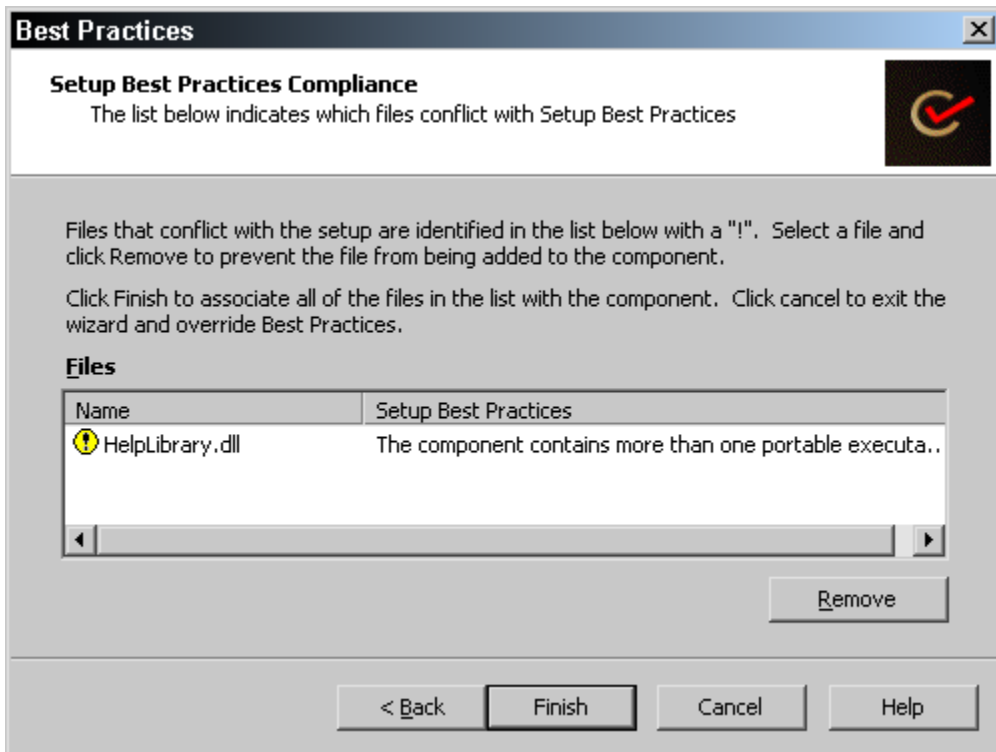


Figure 5-26: *The Setup Best Practices Compliance dialog.*

This means that future setup best practices conflicts will not launch this wizard. Regardless of what action you take in the Welcome panel, the Setup Best Practices Compliance panel appears when you click Next (Figure 5-26).

The Setup Best Practices Compliance panel displays the file or files that are causing the setup best practices conflict. In this case, it shows that there is already a file with a .dll, .exe, or .ocx extension in the component. It states that trying to add HelpLibrary.dll to this component is breaking the rule that says that there cannot be more than one of these types of files in the same component.

You can select the indicated file and remove it so that the rules are not broken or you can click Finish or Cancel and proceed to break the rule. As stated in Chapter 3, these rules are no more than strong guidelines and if you know what you are doing, you can decide to break them.

You have now completed all the operations that are required as part of Step 2 in the View List. In Step 3, you can define the shortcut that is needed for the Developer Art application.

Configure the Target System (Step 3)

Under Step 3 the only action that you need to take is to create a shortcut for the Developer Art application. The Registry, INI File Changes, and Environment Variables views are covered in Chapter 10. The ODBC Resources view is discussed in Chapter 14.

To create a shortcut, go to the Shortcuts/Folders view (Figure 5-27). In this view, by default, you can see all the shortcuts and folders that have been created in the entire project. You can filter what is shown in this view by making a different selection in the View Filter drop-down combo box at the top of this view. For our purposes you can leave the default filter selection so that you can see all the shortcuts and folders in the project.

For the Developer Art application, you need to create one shortcut on the Programs Menu. Because there is only one shortcut, you do not need to create a folder for this single shortcut. Creating a folder for a single shortcut is strongly discouraged by the "Certified for Windows" logo specification. To create a shortcut, right-click on Programs Menu in the sub-view list and select New Shortcut from the context menu.

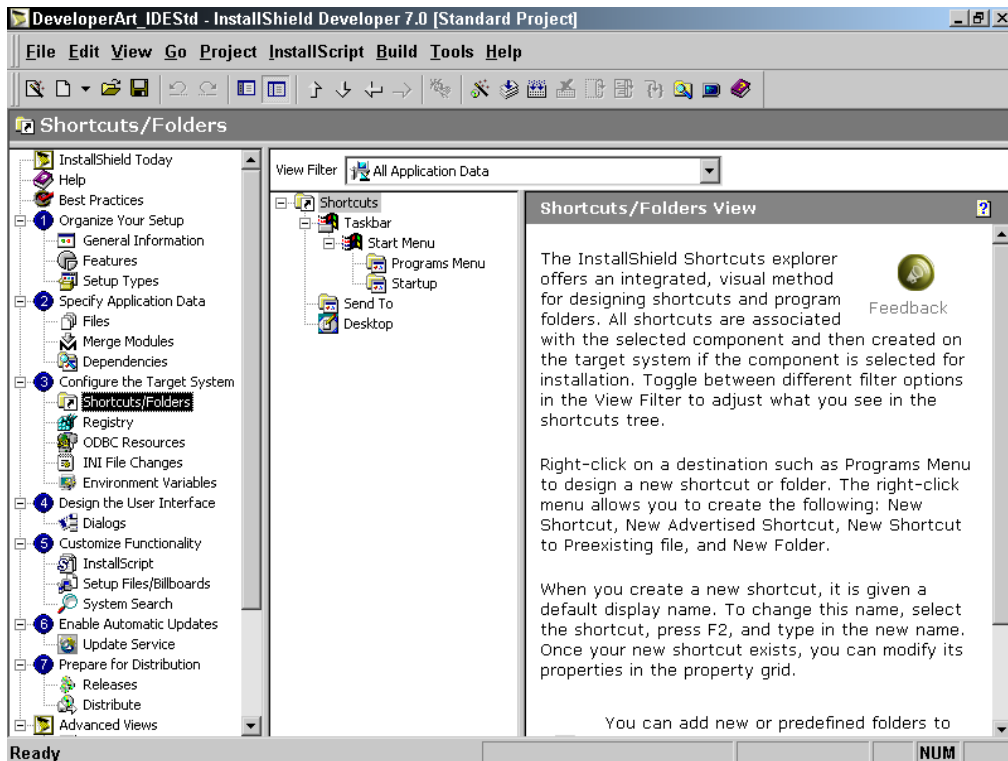


Figure 5-27: *The Shortcuts/Folders view.*

When you do this you will get the Browse for shortcut Target dialog just as you did in Chapter 2. After you have selected DeveloperArt.exe as the target of the shortcut, enter the name of the shortcut as shown in Figure 5-28. The name shown in Figure 5-28 is Developer Art Standard. This entry for the name of the shortcut also sets the default value for the name that will be used on the Start\Programs menu.

The shortcut has a number of properties that we need discuss. Figure 5-28 shows the Shortcut properties as changed for the Developer Art application.

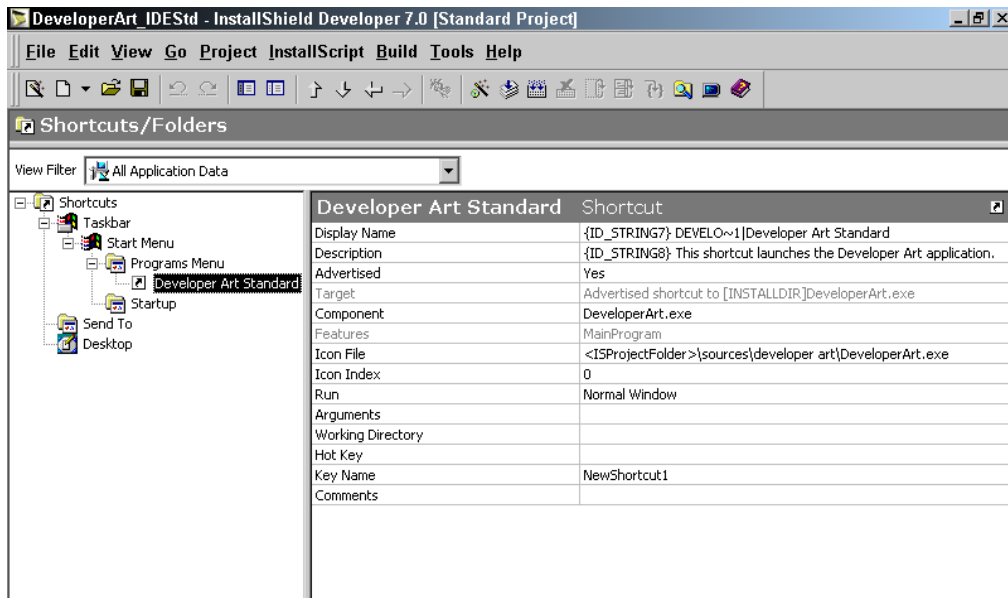


Figure 5-28: *The property panel for the Developer Art application shortcut.*

The following properties are available for shortcuts.

Display Name: The name used for this property is what will be displayed on the Programs Menu. When you create a shortcut, this property is given the same value that you used to define the shortcut. There is no need to make any changes for this property from the name that was used to create the shortcut. InstallShield Developer provides a string that contains both a short file name as well as the name that was entered. The value of this property is used to populate the Name column in the Shortcut table.

The data required for the Name column is of the Filename type. What you enter here is the file name of the shortcut that will be created by the Windows Installer at run time. Because the string "Developer Art Standard" contains more than eight characters and it contains spaces, the value entered into the Name column of the Shortcut table has to contain both the short file name as well as the long file name. The two file names are separated by the pipe symbol (|). If a name of eight or less characters and without any spaces were entered, then the value for this property would consist of just the single string that was entered.

Description: This property contains a string that you want either displayed in Windows Explorer when the end user clicks on the shortcut, or displayed as a tool tip on the Programs Menu when the end user places the mouse pointer over the shortcut. Showing the description of a shortcut as a tool tip on the Programs Menu is available on Windows 2000, but not on any earlier operating system. For the Developer Art application, use the following string value:

"This shortcut launches the Developer Art application."

Advertised: This property is not directly used to populate the shortcut table. It only defines whether you want a shortcut created that can be advertised. If you choose the default value of Yes, then the component identified in the last property has to have a key path that is a file in the component. The file that is being used as the key path for the component will be displayed in the value field of the Target property. If you select No for this property then you would need to define a path to the file or folder in the value field of the Target property that will serve as the target of the shortcut to be created. As you remember from Chapter 2 the default shortcut that is created is one that cannot be advertised. For this example you should select Yes for this property so that the shortcut can be advertised.

Target: The value for this property is used to populate the Target column of the Shortcut table. If the Advertised property is set to Yes then the value that will be placed in the Target column of the shortcut table will be a foreign key to the Feature table. The row in the Feature table that is identified by this foreign key is the feature that contains the component installing the target of the shortcut. If the Advertised property is set to No then the value for this property is a formatted text string that evaluates to an absolute path to a file or a folder that will be launched by the shortcut. Note that since you have selected to make this shortcut advertisable you cannot edit this property.

Component: In this property, you define the component that will install this shortcut. Click in the field to display an ellipsis button. Clicking this button displays the Browse for a Component dialog, which lists of all the components either by feature or all components without regard to feature association (Figure 5-29).

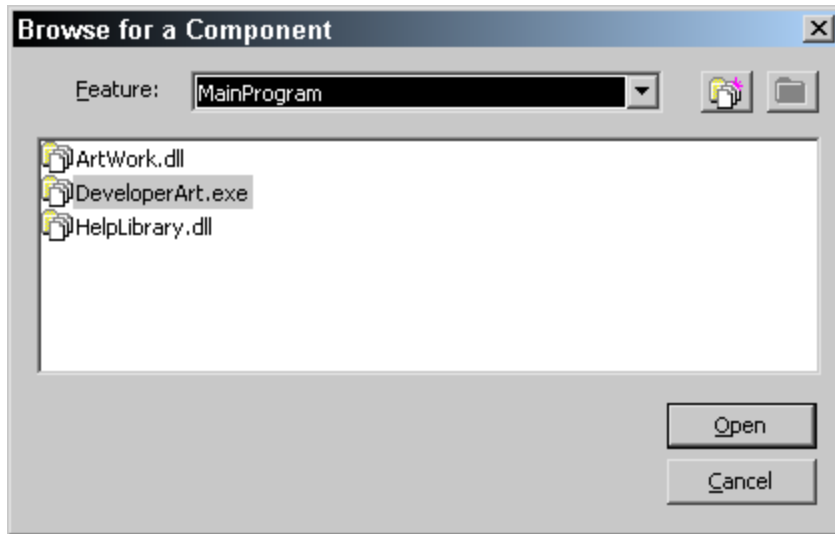


Figure 5-29: *The Browse for a Component dialog.*

The selection you make here has an impact on the value of other properties for the shortcut, as already discussed. You should identify the component before we enter any of the other property values. The selection of component here is used to populate the `Component_` column of the Shortcut table.

Features: This property identifies all the features that install the component that contains the target of the shortcut. This is a read-only property.

Icon File: This property identifies the file that will be used to provide the shortcut's icon. Click in this field to display the ellipsis button that allows you to browse to a file. When you select a file, the Change Icon dialog displays the icons available in the file. You can select the icon to use and click OK. The index for the icon is placed as the value of the Icon Index property. During the build process InstallShield Developer extracts the indicated icon and places it into a resource file that is then streamed into the Icon table. In the Shortcut table, an entry is made in the `Icon_` column and this entry is a foreign key into the Icon table. By default the first icon in the file used as the target of the shortcut is selected. You can change this default by either selecting another icon in this same file or browsing to another file and selecting an icon to be used.

Icon Index: The value of this property is used to extract the icon from the file identified in the Icon File property. Normally this value is populated when you browse to the file that is designated to provide the icon for the shortcut. The first icon in a file always has an index of 0. This value is not used to populate the IconIndex column of the Shortcut table. The value used to populate the IconIndex column of the Shortcut table depends on the index of the icon after it has been placed into the resource file that is streamed into the Icon table.

Run: The value of this property specifies how the application is to be launched when the shortcut is activated. The possible values for this property are Normal Window, Maximized Window, or Minimized Window. The value selected for this property is used to populate the ShowCmd column of the Shortcut table. The value that is placed in the ShowCmd column is the integer 1 for normal window, 3 for a maximized window, or 7 for a minimized window.

Arguments: This property specifies any arguments that need to be passed to the file that is the shortcut's target. For the Developer Art application there are no arguments that need to be passed, so leave this value NULL. Any value defined for this property is used to populate the Arguments column of the Shortcut table.

Working Directory: This property defines the location in which the shortcut starts. The value that is placed here is usually the location where the application should default for the Save as and the Open dialogs that are launched from the File drop-down menu of an application. This value needs to be an identifier into the Directory table. The identifier needs to resolve to an absolute location on the target machine. Click in the field for this property to display the drop-down menu that provides a selection of the available identifiers. You can also choose to create a new identifier by selecting the "Browse, create, or modify a directory entry" option. This launches the Browse for Directory dialog as discussed in Chapter 2. The entry made here is used to populate the WkDir column of the Shortcut table.

Hot Key: The hot key for a shortcut is a mechanism provided by Windows that launches a shortcut through a combination of keyboard keys. It is recommended that installations do not set a hot key for any shortcuts that are being created by the installation. Doing this can conflict with hot keys that are already enabled on the target machine. The setting of hot keys for shortcuts should be left to the end user after the application has been installed.

Key Name: The value of this property is the identifier that is used as the primary key in the Shortcut table. You can edit this value if you need to access the Shortcut table at run-time using a custom action. By default the identifier placed in this property uses the form NewShortcutX where X is a sequence number. If you plan to create your own value for this property it must follow the rules laid down by Microsoft for the identifier data type and you cannot use this same value for two different shortcuts because of the uniqueness requirements of a primary key.

Comments: Use the Comments property to provide information about the shortcut's purpose for anyone working on this project. These comments are saved in the project file and are not used in the MSI database.

You have done what is necessary to create the shortcut for the Developer Art application. The operations that can be performed under Step 4, Step 5, and Step 6 are not required for completing the installation program for the Developer Art application. Before we move on to discussing the creation of a build we need to take a tour of the views under Advanced Views at the bottom of the View List.

Advanced Views

The views under the Advanced Views list, Figure 5-30, provide functionality that is not available anywhere else in the IDE. Some of these views provide duplicate functionality that is presented in a different format. We will discuss some of these views in detail and some will only be introduced.

Path Variables

Click on Path Variables to display the Path Variables view, which allows you to define path variables (Figure 5-31). Path variables are used to locate the source files for the application when performing a build. In other words, a path variable is a build-time functionality only and has nothing to do with locating files during the running of an installation. You saw in Chapter 3 that the Directory table is used to provide the source and destination locations for copying files during the running of the installation program. Chapter 2 provided an introduction to the subject of path variables.

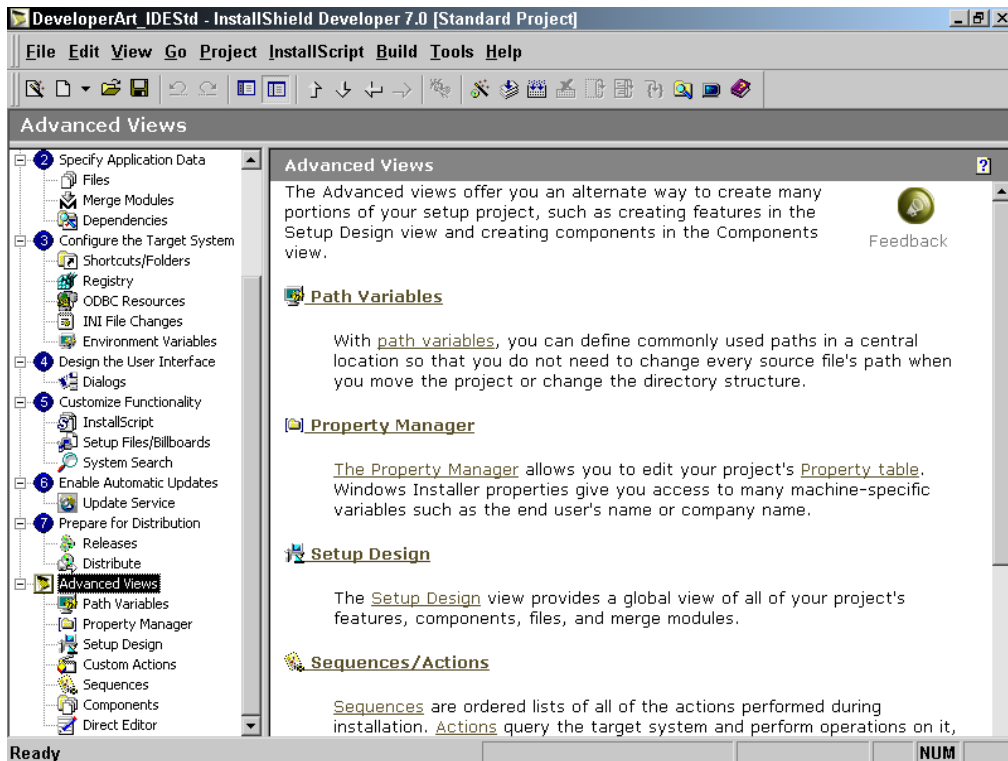


Figure 5-30: *Advanced Views* list of views in the IDE.

The value of a path variable is that it makes it very easy to relocate or redefine the structure of the source files for an application. If you define the location for the source files with absolute paths and wanted to change the location of some or all the files on the build machine, you would have to re-link each of these files by going to each component in the project, deleting the present link, and re-adding the files to create a new link. If you use path variables, all you have to do is go to the Path Variables view and change the location to which the path variables point. The one thing that you cannot do is, after using a particular path variable to link to files, change the name of the path variable. If you do this, you have to re-add all the files that used the path variable's original name.

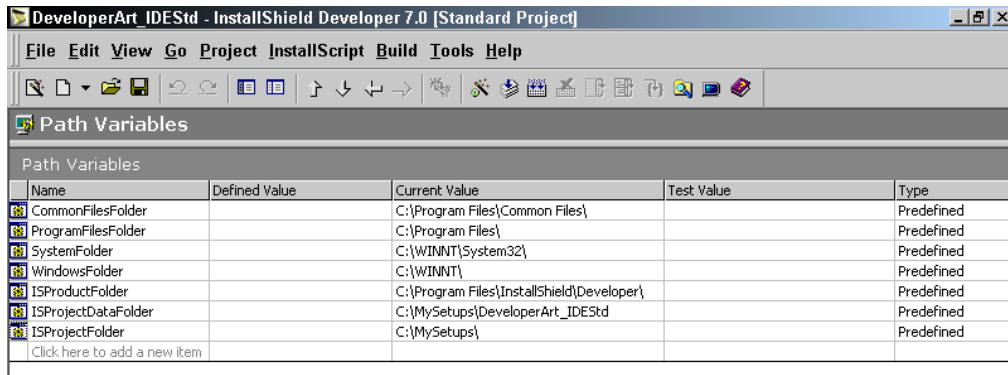


Figure 5-31: *The editing panel for creating or modifying path variables.*

There are five columns in which the parameters of a path variable are displayed (Figure 5-31). The Type column defines the type of path variable that is being displayed in the row. There are four types of path variables: Predefined, Standard, Environment, and Registry. Predefined path variables are set by InstallShield Developer and cannot be modified. You cannot create your own Predefined path variables. Note that the Predefined path variables point to common locations on the build system such as the folder where Windows is installed and where InstallShield Developer is installed. Of particular interest is the path variable named ISProjectFolder and it points to the location you specify in the Options dialog for where projects are to be created. Also the ISProjectDataFolder path variable points at the root location for all the builds created for this particular project.

If you want to create a Standard type of path variable, click in a new row of the Name column. A new Standard path variable is created with a default name. The Current Value column shows that this value is presently undefined. As an experiment, type into the Defined Value column the location where the Developer Art application source files can be found. Once you have done this you will see that the value in the Current Value column has the same value as you just entered. Give the path variable a better name, such as StdSources instead of NewCustomPathVariable1.

When you create a new path variable when adding files to a component, you are creating a Standard path variable. A Standard type of path variable always shows the same values in the Defined Value column and in the Current Value column. As seen in Figure 5-31, the Predefined path variables have a value in the Current Value

column and nothing in the Defined Value column. The Current Value column is a read-only column.

If you want to create an Environment type of path variable, click in a new row of the Type column and select Environment from the drop-down menu. A new path variable is created with a default name. The Current Value column shows that this value is presently undefined. As an experiment, type into the Defined Value column the name of an environment variable such as TEMP. Give the path variable a better name, such as EnvSources instead of NewCustomPathVariable1.

You can also experiment to see how the Registry type of path variable works. You can reference either the default value of a registry key or the data for a registry value for the location at which this type of path variable can point. To experiment with this, there is a key in the registry that was created when you installed InstallShield Developer. This key is:

```
HKEY_LOCAL_MACHINE\SOFTWARE\InstallShield\Developer\7.0
```

Under this key there is a value name that has as its data the installation location for InstallShield Developer. The value name and value data are:

```
Install Location=C:\Program Files\InstallShield\Developer\
```

To create a Registry type of path variable, click in a new row of the Type column and select Registry from the drop-down menu.

Enter the following in the Defined Value column:

```
HKEY_LOCAL_MACHINE\SOFTWARE\InstallShield\Developer\7.0\Install
                                     Location
```

When you make this entry in the Defined Value column, the Current Value column displays the value data of the Install Location value name as follows:

```
C:\Program Files\InstallShield\Developer\
```

You should also provide a better name for the path variable, such as RegSources.

When you have made these entries for both the Environment and Registry types of path variables, the Path Variables view should look similar to what is shown in Figure 5-32.

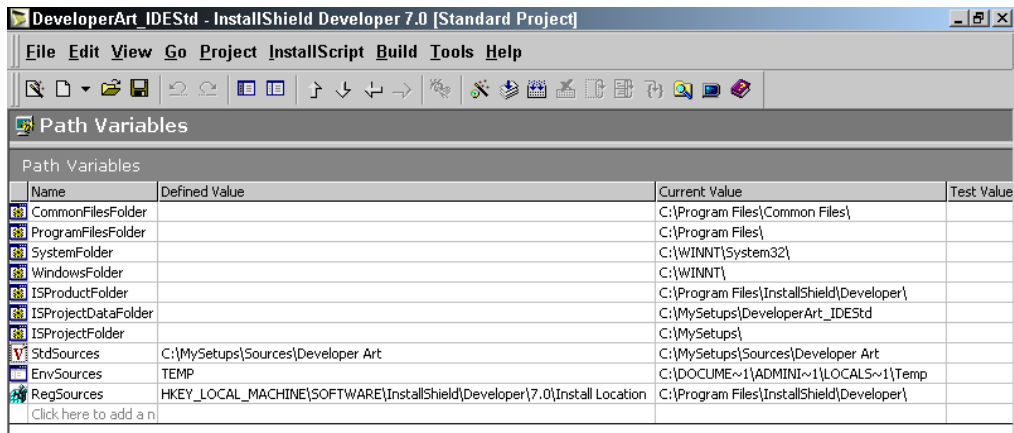


Figure 5-32: *The Path Variables view with Standard, Environment, and Registry types.*

For the Environment type of path variable, the environment variable has to already exist when InstallShield Developer is launched or it will not be recognized. If you launch InstallShield Developer and then create the environment variable, you have to reboot InstallShield Developer before the environment variable is recognized. If you use an environment variable that is defined for both the current user as well as the system, then the value used for the path variable will be the value defined for the current user. If there is an environment variable defined only for the system then the value of this variable will be used for the value of the path variable.

If `Install Location` were a sub-key and not a value name then the value shown in the Current Value column would be the default value for the key. If however, there were both a sub-key named `Install Location` and a value name called `Install Location` under the following registry key then the data for the value name would be used instead of the default data value for the key.

```
HKEY_LOCAL_MACHINE\SOFTWARE\InstallShield\Developer\7.0
```

This situation is shown in Figure 5-33. If you make the following entry into the Defined Value column as before, the data value for the value name `Install Location` will take precedence to the default data value for the `Install Location` key.

```
HKEY_LOCAL_MACHINE\SOFTWARE\InstallShield\Developer\7.0\Install
Location
```

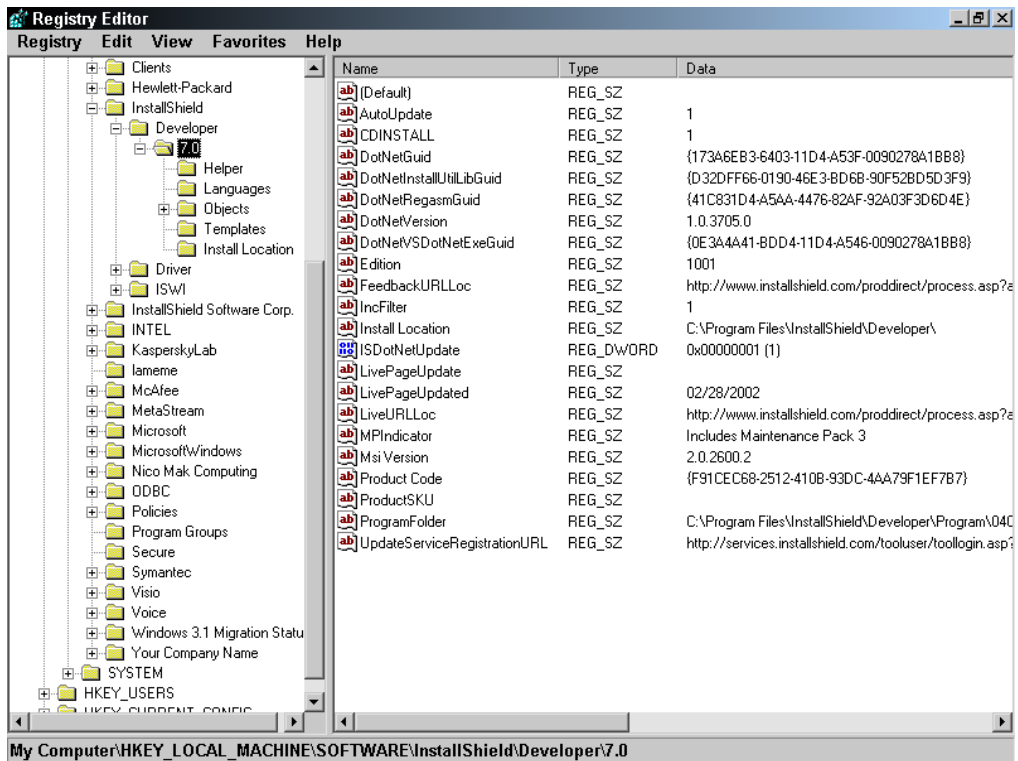



Figure 5-33: *The registry where there is both a value and a key with the same name.*

The last column to be discussed is the Test Value column. This column provides a hard-coded path as a backup location to the location that is defined in the Current Value column. This is to provide the capability to still build a project if the primary location for the application's source files is unavailable. This could happen if the primary location for accessing the source files during a build is a network location that for one reason or another is not available when a build has to be made.

You can remove these example path variables by right clicking and selecting the delete option.

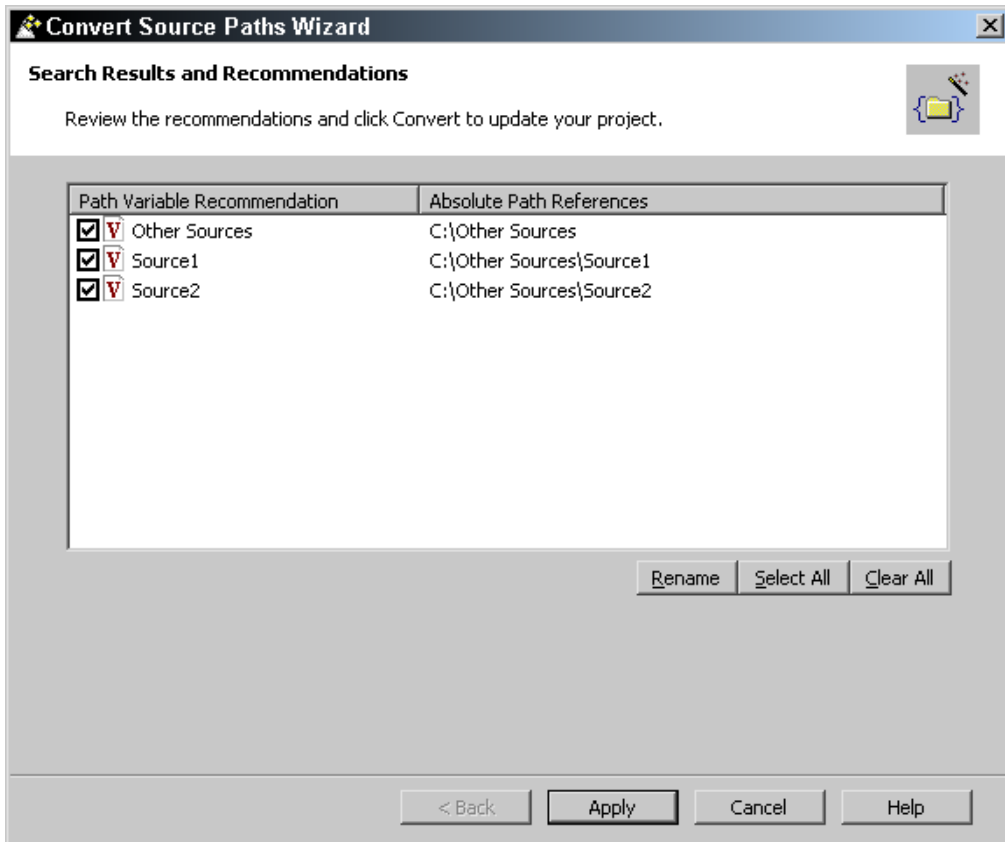


Figure 5-34: *The Search Results and Recommendations dialog in the Convert Source Paths Wizard.*

If you have used absolute paths in your project to link to source files in your components, you can scan the project and change these links to path variables. You can do this by running the Convert Source Paths Wizard from the Convert Source Paths option on the Project drop-down menu. When the Convert Source Paths Wizard is launched, the first panel prompts you to search by clicking on the Search button. When the search is complete, the wizard displays the Search Results and Recommendations dialog showing all the absolute paths that were found along with recommendations for the path variables (Figure 5-34).

The Search Results and Recommendations panel provides the option of accepting the recommended path variable names or renaming them to something different. You

can also decide not to create path variables for certain locations. When you have made any changes to the path variable names, click Apply to convert the absolute paths to path variables. If you perform a search that does not find any absolute paths being used to link files, then the final panel of the wizard appears. Dismiss the panel by clicking Finish.

Property Manager

As discussed in Chapter 3, one of the important tables in the database in the MSI file is the Property table. The Property Manager view provides you with the facility to create custom properties or to modify the values of existing properties (Figure 5-35). There are three columns in this view: Name, Value, and Comments. The Comments column is only for internal project use and entries in this column are not built into the MSI database.

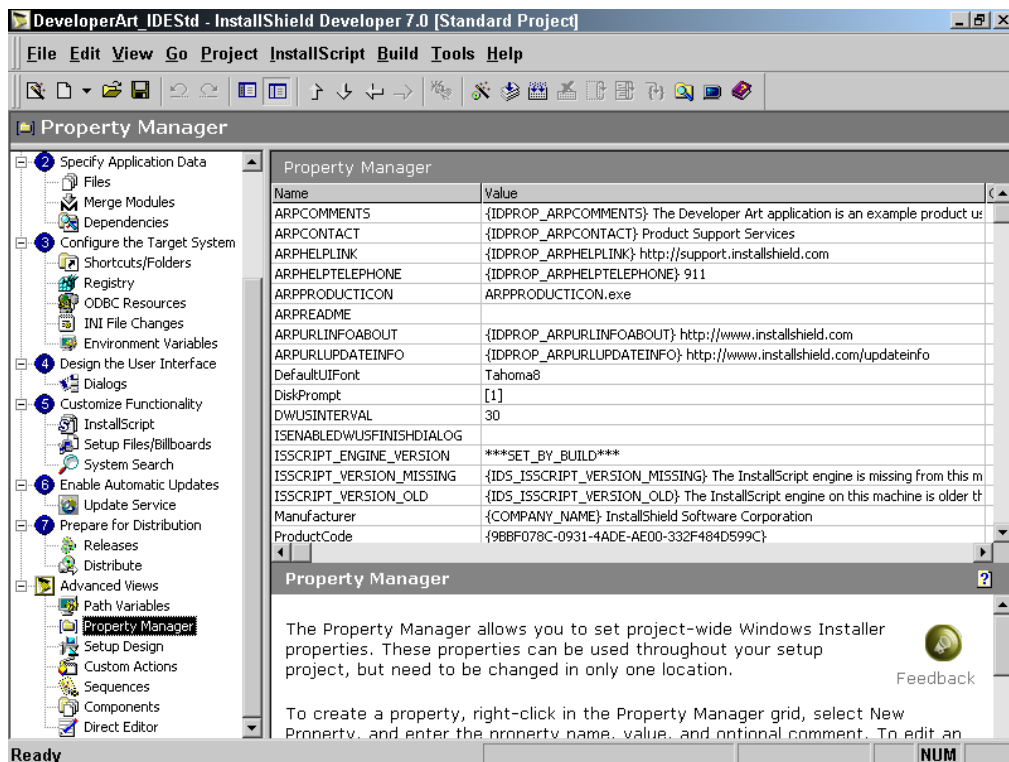


Figure 5-35: The Property Manager view.

Right click in the Property Manager view to see the context menu. The options provided are New Property, New Localizable Property, Make Localizable, Clear Property, Delete Property, and Delete Property & String Entry. When you choose New Property, a new property is created with a default name and a default value of 0. You can left-click twice in the Name column and change the name of the property and can left-click twice in the Value column and enter the value that you want for this property. To delete a property and its value, position the mouse pointer over the row that you want to delete. Right click and select Delete Property from the context menu.

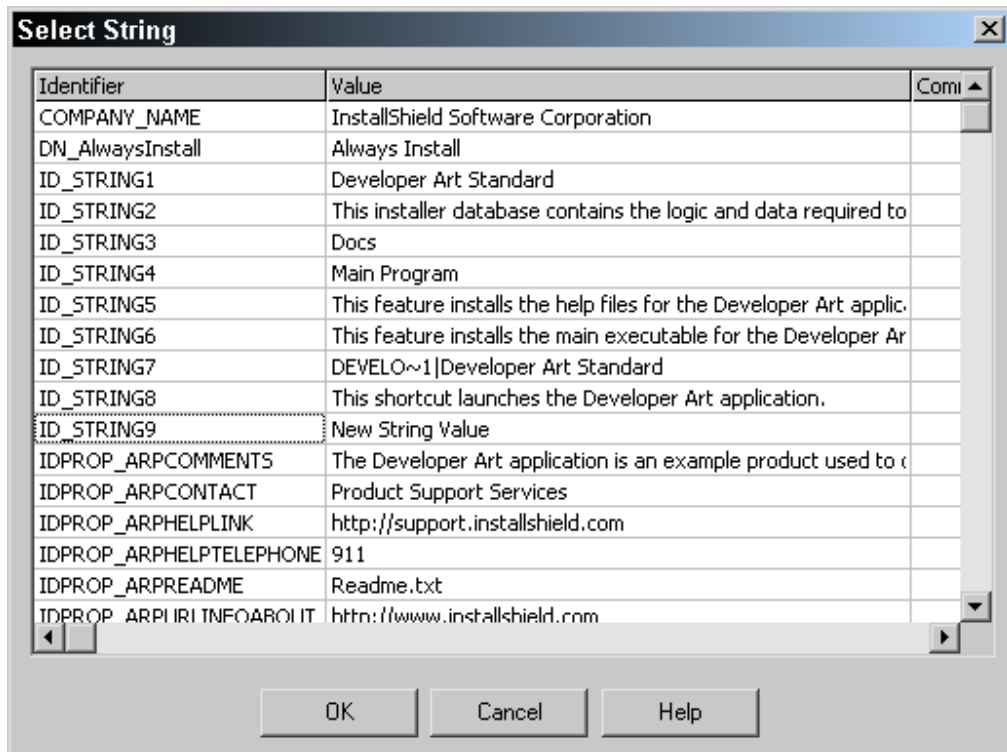


Figure 5-36: *The Select String dialog for the Property Manager.*

If you want to create a property that has a link into the string table, right click and select New Localizable Property. This creates a default property name and a default value string that is identified by a default string ID that is shown inside curly braces ({}). Left click twice in the Name column and change the name of the property. In

the Value column, when you left-click twice, the ellipsis button appears. Clicking this button launches the Select String dialog (Figure 5-36).

In this dialog, you can rename the default string ID and you can enter the value that you want for the property. You can also right-click and add a new entry in the string table or delete an existing string table entry. To select an existing string and bring it into the Property Manager, left click in the appropriate row and click OK. This will then bring in the selected string value and string ID into the Value column of the Property Manager. To delete the property from the Property Manager, right click on the row to be deleted and select Delete Property. To delete both the property from the Property Manager and the entry in the string table, right-click on the row and select Delete Property & String Entry.

Setup Design

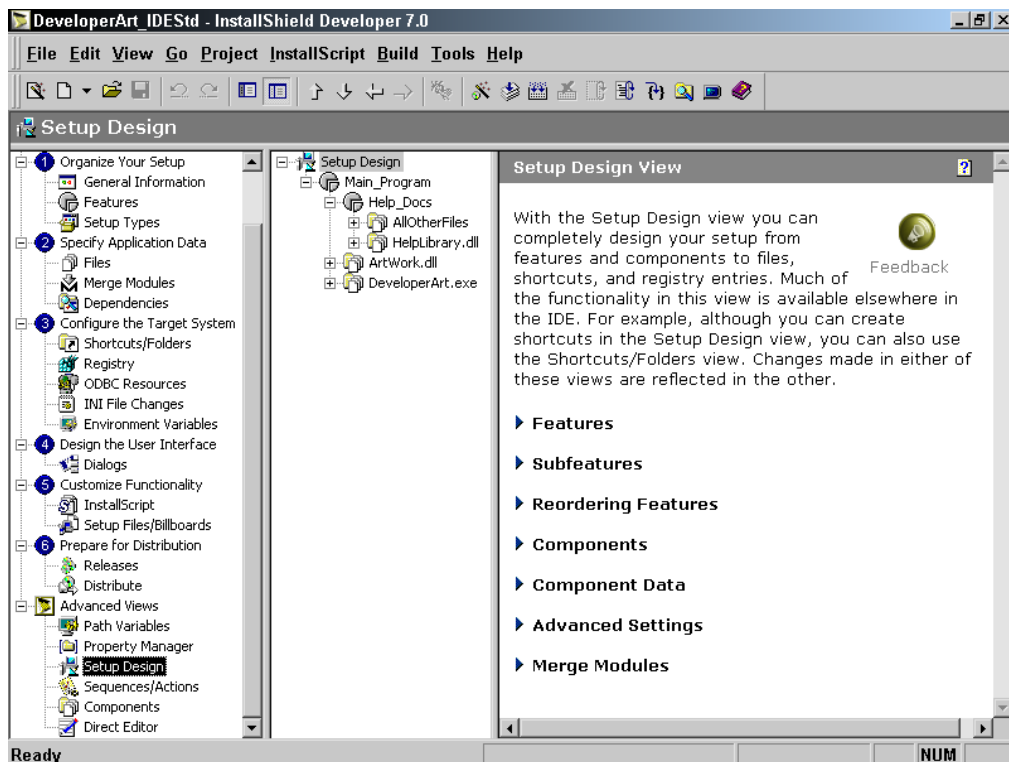


Figure 5-37: The Setup Design view for the Developer Art application.

The Setup Design view provides a different way to look at the structure of your setup project. The Files view shows the components that compose your project listed according to the destination to which they will be copied during the installation. In the Setup Design view, you can see your components according to the features with which they are associated.

Click on the Setup Design view in the View List and expand the tree under each of the features in the sub-view list to see what is shown in Figure 5-37.

If you right-click on Setup Design in the sub-view list the context menu provides an option to add a new feature to the setup. If you right click on a feature, the context menu that is shown here appears.

New Feature	Ins
New Component	
Insert Components...	
Delete	Del
Rename	F2
Move Up	Ctrl+Shift+Up
Move Down	Ctrl+Shift+Down
Move Left	Ctrl+Shift+Left
Move Right	Ctrl+Shift+Right
Component Wizard...	

From this context menu you can also create new features as sub-features of the feature from which you launched the context menu. You can also create new components under this feature. Choosing the New Component option or the Component Wizard option can create new components. The Component Wizard option is used to create special types of components, which are discussed in Chapter 14. You can also delete features,

rename them, and adjust their relationship with other features such as making sub-features into top-level features.

Deleting a feature does not delete any of the associated components. The components remain in the project, but they are not built into the database if they are not associated with a feature. They can be considered a pool of components ready for use when you want to associate them with a feature. To associate a component with a feature, select Insert Components from the context menu. When you do this, the Insert Components dialog appears (Figure 5-38). This dialog lists all of the components defined in the project file that have not already been associated with the selected feature.

When you select a feature in the Setup Design view, the feature's properties panel is displayed. This is the same properties panel that is displayed in the Features view. You can set all of a feature's properties in this view just as well as in the Features view. If you select a component, you will see a different properties panel that displays the properties and their values that are pertinent to the selected component. We will briefly discuss these properties below when we talk about the Components view.

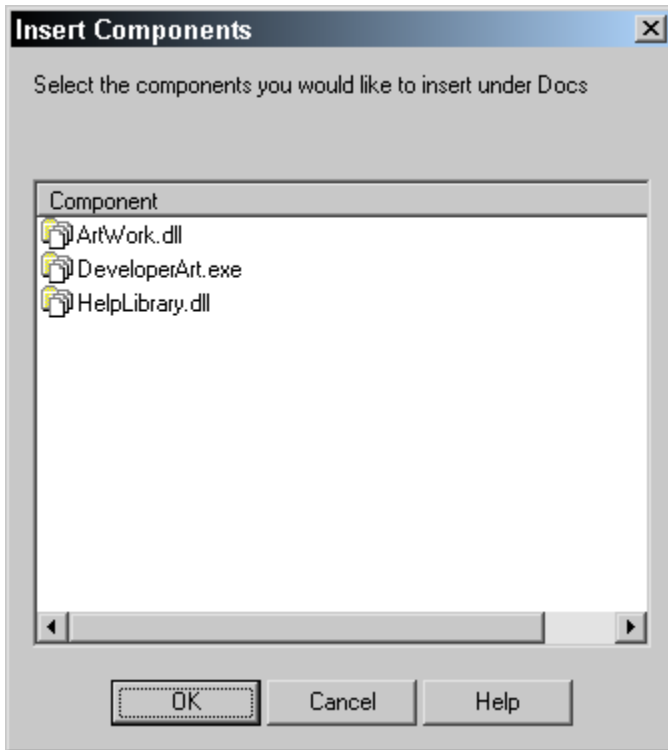


Figure 5-38: *The Insert Components dialog for the Docs feature.*

Custom Actions

You use this view to create the custom actions that you want to have in your project. Since you have not created any custom actions this view is not very interesting. In Chapter 4 you saw that there are a number of built-in custom actions that InstallShield Developer brings into play whenever a project uses InstallScript. This of course is the case with a Standard project or a Basic MSI project that uses

InstallScript custom actions. These built-in custom actions do not appear in the Custom Actions view since you are not supposed to edit them. However, they can be seen in the Direct Editor view. Custom actions are discussed in Chapter 3. Chapter 11 provides a complete discussion on the creation of InstallScript custom actions.

Sequences

The Sequences view is where you can control the sequence of operations that occur during an installation (Figure 5-39). In Chapter 3, we learned that an installation is made up of actions that are placed in sequence tables in order to define when the actions are to be executed. In the Sequences view, you can see the order of actions and you can add, remove, and reorder the actions in the sequence tables of the database.

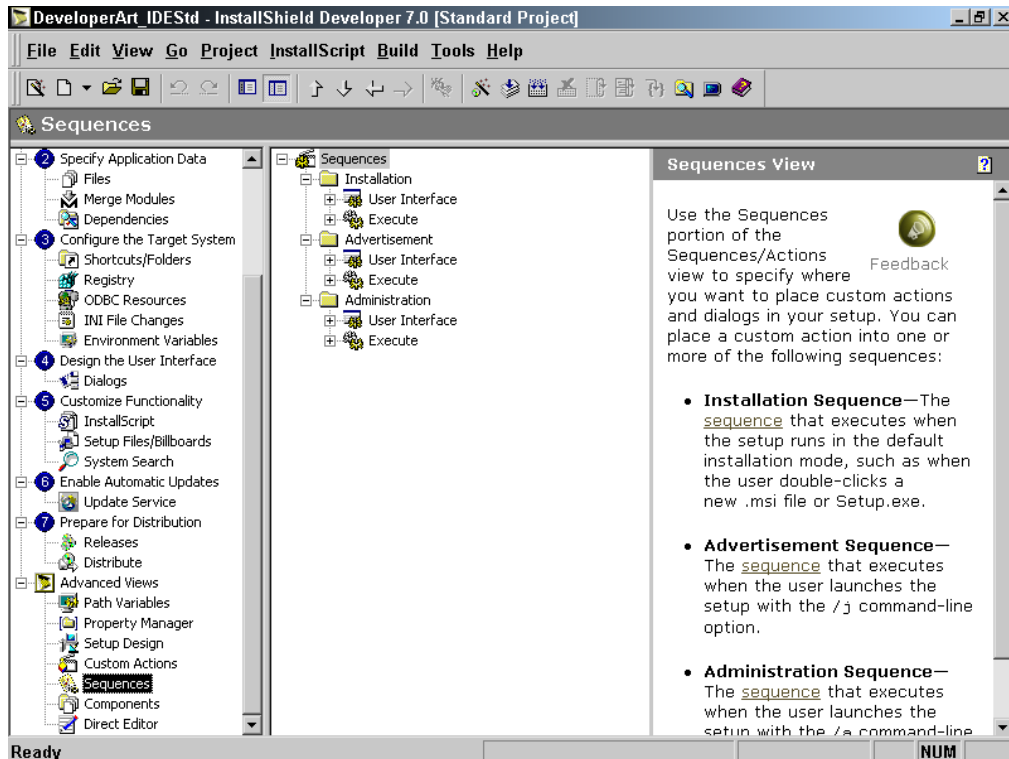


Figure 5-39: The Sequences view for the Developer Art application.

The Sequences view contains the Sequences node. Under the Sequences node there are three folders named Installation, Advertisement, and Administration. These three folders correspond to the three top-level actions that were discussed in Chapter 3. Under each of these folders there is a User Interface and an Execute node. Under each node is a list of the standard actions and custom actions that are in the respective user interface and execute sequence tables. These actions are displayed in the order of their sequence numbers from lowest to highest.

Expand the list of actions under either the User Interface or Execute headings to see the default list of actions for a Standard project. Right-click on any of these actions to display a context menu that gives you the ability to insert new actions, remove actions, and change the location of the actions in the sequence table.

Components

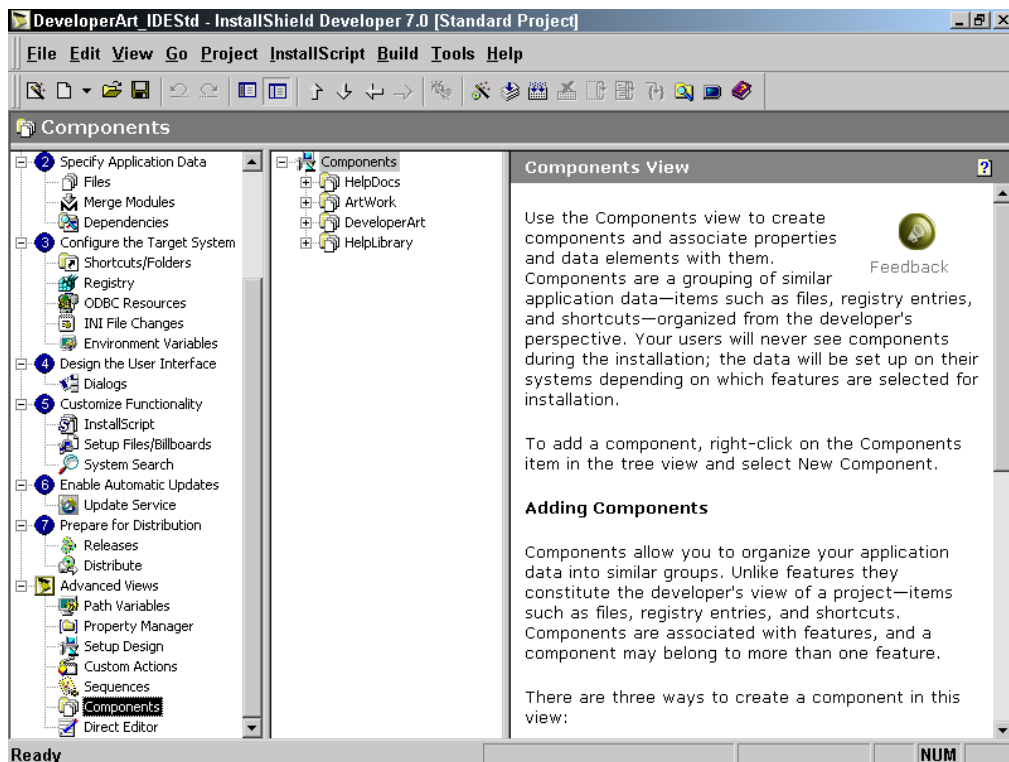


Figure 5-40: The Components view for the Developer Art application.

The Setup Design view shows your project's features and the associated components. In the Components view, Figure 5-40, you see all of the components in the project whether they are associated with a feature or not.

In Components view, right-click on the top-level heading to display a context menu that allows two approaches to creating new components. Note, however, that when you create components in this view, they will not be associated with a feature. To associate a component created in this view with a feature, you need to go to the Setup Design view and insert the component under the appropriate feature. When you right click on a specific component you get a context menu that allows you to delete it from the project, rename it, or export the component to another project. If you want your project to look the same as the one on the CD-ROM at the back of the book you can rename each of the components in your project as shown in Figure 5-40.

COMPONENT PROPERTIES

As already mentioned, each component has a set of modifiable properties. When you create a project, these properties receive default values but it may be necessary to change many of these default values for a real world project.

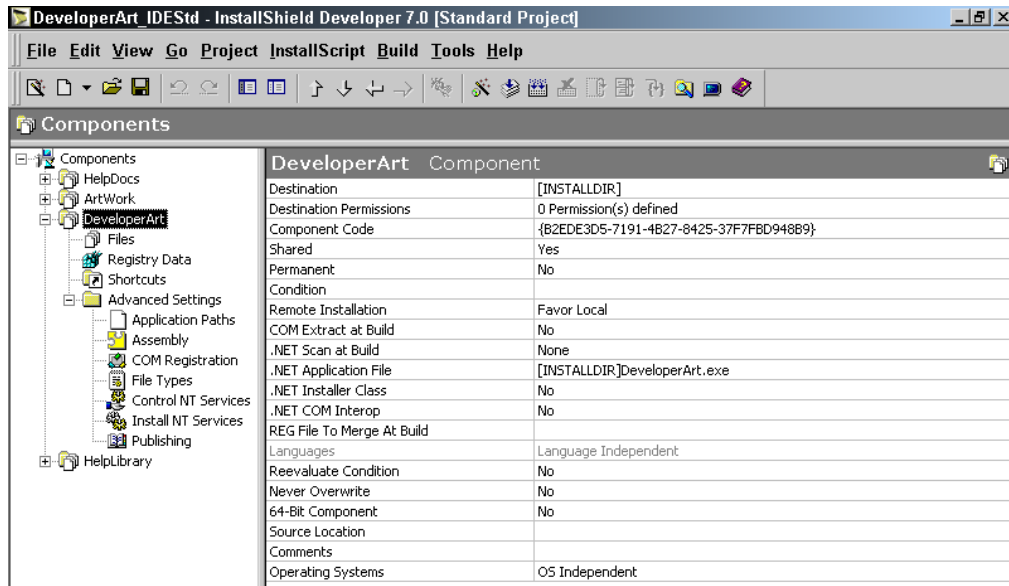


Figure 5-41: The DeveloperArt.exe component property panel.

This section briefly discusses these properties. We will return to these properties in Chapters 13 and 14 to discuss some of them in more detail.

The property panel for the DeveloperArt.exe component is shown in Figure 5-41. Each of these properties is discussed briefly below:

Destination: This property of a component defines where the files in the component will be copied during the installation. This is a required property and it is used to populate the Directory_ column of the Component table. The value for this property can be selected from the drop-down editable combo box that appears when you left click in the field. You can edit this field so as to provide hard coded folder names under an existing Directory table identifier.

Destination Permissions: When you click in the value field for this property you will get an ellipsis button that will launch the Permissions dialog. You use this dialog to author the LockPermissions table. The entries in this table are used to secure individual portions of your application in a locked-down environment. The security concerns related to Windows NT, Windows 2000, or Windows XP are beyond the scope of this book.

Component Code: Whenever you create a new component, InstallShield Developer generates a GUID that is used as the value for this property. This value is used to populate the ComponentId column in the Component table and is how the Windows Installer distinguishes one component from another. This is a very important value and there will be many times that you do not want to use the value that InstallShield Developer gives us. There are a number of guidelines from Microsoft about the use of component codes and when they should be changed if you are upgrading the component. The importance of this property is discussed in Chapter 13.

Shared: Legacy applications and applications that use the Windows Installer for their installation use different methods for keeping track of the number of applications that have installed the same file. This property links the two approaches. To be on the safe side the value of this property is set to Yes by default. This is what you should do for the Developer Art application. The value for this property is used as one of the bit-flags that make up the value placed in the Attributes column of the Component table.

Permanent: If you want a component to be installed but never uninstalled, set this property to Yes. In the case of the Developer Art application you do not want to make any of the components permanent. The value for this property is used as one of the bit-flags that make up the value placed in the Attributes column of the Component table.

Condition: There may be times when you want a component to be installed only under certain conditions. When that is the case, you enter a condition statement as the value of this property. When it evaluates to TRUE, the component is installed and when it evaluates to FALSE, the component is not installed. A NULL value for this property is the same as a condition that is always TRUE. The value of this property is used to populate the Condition column of the Component table.

Remote Installation: This property is similar to the same property for features. When you left-click in the value field for this property, a drop-down menu provides three possible selections: Favor Local, Favor Source, and Optional. Select the Favor Local option if the component needs to be copied to the local hard drive in order for it to run. Select the Favor Source option if the component will not be copied to the target machine, but instead will be left on the distribution media and run from there. Select Optional if the component will either be copied to the local hard drive or left on the distribution media depending on the selection of the associated feature's install state. A feature's install state has a default value that can be changed in the custom setup dialog normally presented in the user interface of an installation program. The value for this property is used as one of the bit-flags that make up the value placed in the Attributes column of the Component table.

COM Extract at Build: When you have a COM component, you can opt to have the associated registry values extracted from the component every time the project is built. These COM registry values are not saved permanently in the project file and using this approach is appropriate when the COM server is still under development and the registration values may change. When a COM server is not undergoing changes, it is more efficient to extract the COM registration information into the project file so that the build process can proceed much faster. In this case you would use the Component Wizard to create this component. This wizard is discussed in Chapters 13 and 14. It should be noted that, although the ArtWork.dll component is a COM server, you do not have to take any special action to have the Extract at Build property set to Yes. When we

added ArtWork.dll to the Main Program feature, InstallShield Developer recognized that this file was a COM server and automatically set this property to Yes. This is a build-time property only.

.NET Scan at Build: Using this property you indicate whether this component is to be scanned for .NET dependencies or properties at build time. The three possibilities that are presented in the drop-down combo box are to perform no scanning, to scan for both dependencies and properties, or to scan only for properties. The installation of .NET components is not covered in this book.

.NET Application File: This property is only used when the component is scanned for .NET dependencies. This property sets the value of the File Application property for a .NET assembly. The installation of .NET components is not covered in this book.

.NET Installer Class: This property is relevant only if the component contains a specific derived class. The installation of .NET components is not covered in this book.

.NET COM Interop: Setting this property to Yes allows a COM object to call a .NET assembly. The installation of .NET components is not covered in this book.

REG File to Merge at Build: Using this property you can identify the name of a .reg file that you want imported every time you make a build. This is valuable if you have registry entries for a component that are constantly changing and you do not want to have these registry values permanently included in your project file.

Languages: This particular property has only build-time use and has nothing to do with populating any column in the Component table. If you want to associate a component with one or more specific languages, left-click in the field to display a list of languages from which you can choose the languages with which to associate the component. When you build the project into an installation program using the Release Wizard, you can filter out components based on their language association. This filtering is discussed when we cover the Release Wizard and it is part of the SKU management functionality available in InstallShield Developer.

Reevaluate Condition: When you place a condition on a component under normal circumstances, it is evaluated only during the initial installation. If the end user subsequently performs a maintenance operation on the installed application, the condition is not reevaluated even if the evaluation of the condition has changed its result. If you want to have a condition reevaluated every time a maintenance operation is performed on the installed application, you need to set the value of this property to Yes. The value for this property is used as one of the bit-flags that make up the value placed in the Attributes column of the Component table.

Never Overwrite: If a component is already registered on the target machine and this property has been set to Yes, the component will not be installed even if it is a later version of the component. The value for this property is used as one of the bit-flags that make up the value placed in the Attributes column of the Component table.

64-bit Component: If the component is a 64-bit component, set this property to Yes so it is properly registered. The value for this property is used as one of the bit-flags that make up the value placed in the Attributes column of the Component table.

Source Location: This property is used to create a distribution image where there are different files that for one reason or another have the same name. In a situation like this, two different files with the same name cannot exist in the same folder on the distribution image. Here you have the capability to name different folders for each of the files that have the same name so that on the distribution media they will not conflict with each other. The entry made here has an impact on the how the Directory table is constructed.

Comments: Just as with features and shortcuts, the Comments property is used to enter information into the project file for your use in remembering any specifics about the component that are important. This information is not placed in the database.

Operating Systems: If a component is specific to any particular set of operating systems, you need to identify that set of operating systems in this property. Do this by left clicking in the field for this property and then clicking the ellipsis button. This launches the Operating Systems dialog, from which you choose the operating systems. If the component is not specific to any operating system, leave

the default value as OS Independent. This property is here to enabling migration from earlier projects created by the InstallShield Professional. This same functionality can be accomplished by creating the correct condition in the Condition property.

COMPONENT SUB-VIEWS

Under each of the components shown in Figure 5-40 is a tree of sub-views in which you can manipulate component data, including indicating what files should be installed by the component and the registry entries that should be created when the component gets installed. The fully expanded tree of sub-views for the DeveloperArt.exe component is shown in Figure 5-41. Here we discuss only the Files, Shortcuts, and the Application Paths sub-views. The other sub-views will be discussed at the appropriate time in Chapters 13 and 14.

Click on the Files sub-view to see a list of all the files that are contained in the component. When you right click in the Files panel, a context menu appears (Figure 5-42). From this context menu, you can add files, remove files, set a file as the key path, and perform other component-related operations.

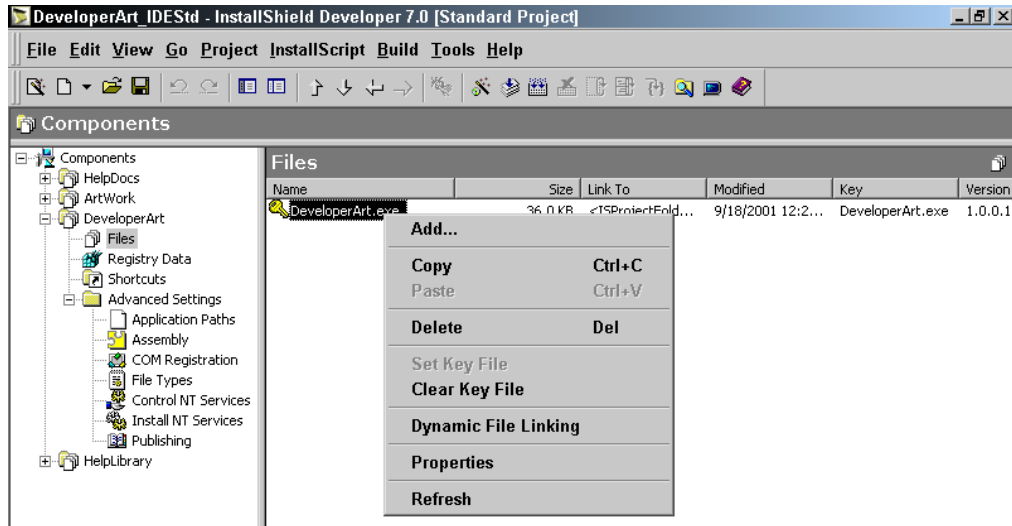


Figure 5-42: *The Files sub-view for the DeveloperArt.exe component.*

We will discuss the Dynamic File Linking option in Chapter 13. The important thing to note is that you can do everything here that you can do in the Files view, so if working here makes more sense to you, then this is where you can construct your installation program.

We now move to the Shortcuts sub-view, which allows you to create shortcuts on a component-by-component basis. When you click on the Shortcuts sub-view for the DeveloperArt component, you see the same set of properties that we worked with in the Shortcuts/Folders view.

Finally we will look at the Application Paths sub-view for the DeveloperArt component (Figure 5-43). This view allows you to easily create one of the standard entries in the registry that all installations should be making. This entry makes an entry under the App Paths key in the registry, which is located as follows:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\
App Paths
```

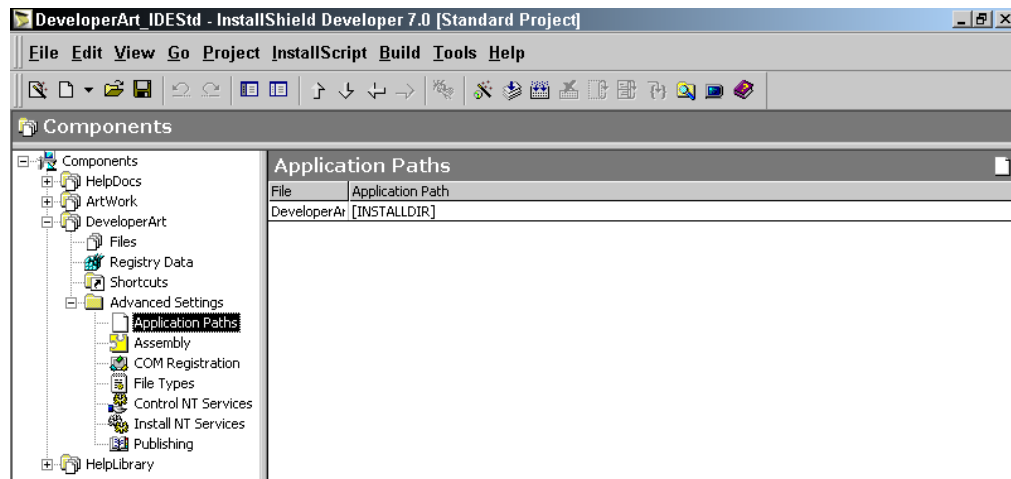


Figure 5-43: *The Application Paths sub-view for the DeveloperArt.exe component.*

Under this key there are sub-keys that are the names of the main executables for the applications that have been installed on the system. These sub-keys normally have two values, the Default and the Path values. The data for the Default value is the absolute path to the executable. This allows you to launch the application from the Run dialog accessed from the Start Menu. In the Run dialog all you have to do is type

in the name of the executable. The more important value is the Path value that has as its data a semi-colon delimited-list of locations where the executable can find the DLLs it needs in order to function correctly.

In the Application Paths sub-view, left click in the File column to select an executable from a drop-down menu. In the Application Path column, another drop-down menu allows you to select the first location for the Path value. If you want to add additional locations to the Path value, you can manually type them in to the Application Path column. For the Developer Art application, select the only executable that is in the component. For the value in the Application Path column, select the [INSTALLDIR] property name.

The last view in the Advanced Views list is the Direct Editor view. This view provides the facility to edit the tables in the database in a fashion similar to that provided by the Orca database-editing tool. This is something that is used in advanced situations and we will discuss its use in the chapters that comprise Part III of this book.

We are now ready to go back to Step 7 and learn about the Release Wizard.

Prepare for Distribution (Step 7)

Under Step 7, we are interested only in the Releases view (Figure 5-44). This is where you will launch the Release Wizard. For this project, you are going to take the basic route through the Release Wizard and we will cover only those features that are required to build the installation program for the Developer Art application.

In Chapter 2 when you used the Project Wizard to make a build, it was using the default selections that are set in the Release Wizard. Having created an installation project directly in the IDE, you could build with the default values by clicking the Build button on the toolbar or selecting the Build option from the Build drop-down menu. What we want to do, however, is use the Release Wizard to create the build for the Developer Art application. Since this is only an introduction to the Release Wizard, here we will discuss only those items that are required to create the distribution image for the Developer Art application.

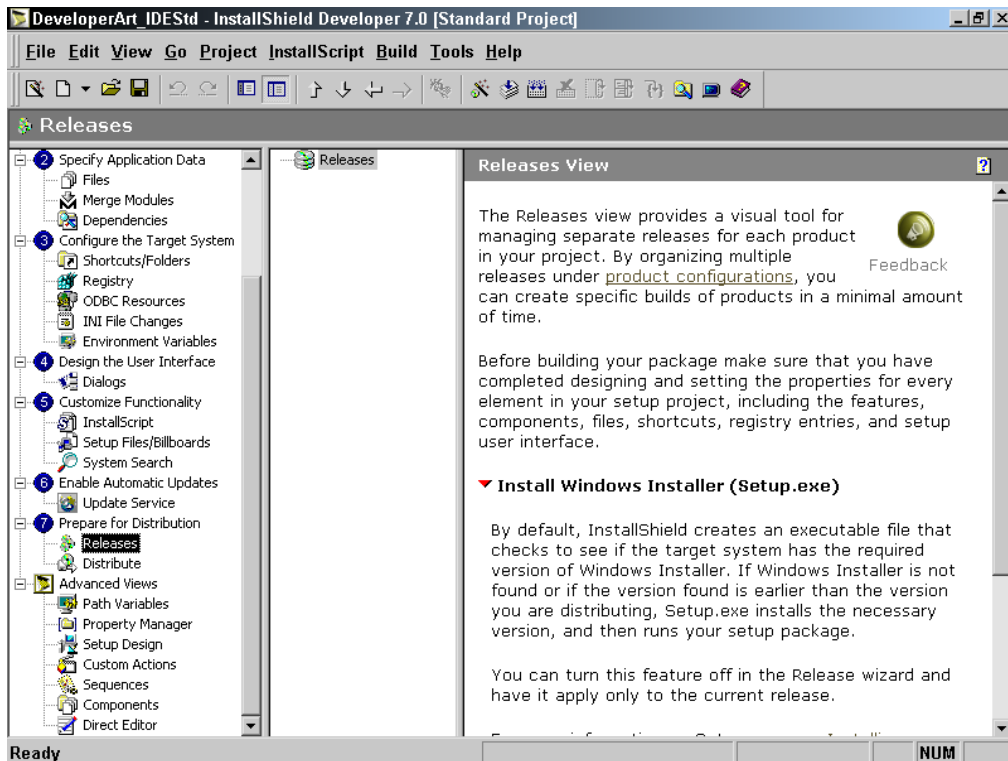


Figure 5-44: *The Releases view.*

You can launch the Release Wizard from the toolbar, from the Build drop-down menu, or by right clicking on the Releases icon in the sub-view list. When you right-click on the Releases icon, a context menu appears with two options: New Product Configuration and Release Wizard. To build the Developer Art project, launch the Release Wizard using one of the above methods. After the Welcome panel appears, click Next to go to the Product Configuration dialog (Figure 5-45).

When you build a project, you are creating a hierarchy of folders where the product configuration is at the top of the hierarchy. Different product configurations define different products that can be built from the same project file. Different products need to have different ProductCode properties and different package code values. Examples of builds that can be made from the same project file would be the installation program for the English version of a product and the installation program for the German version of the same product.

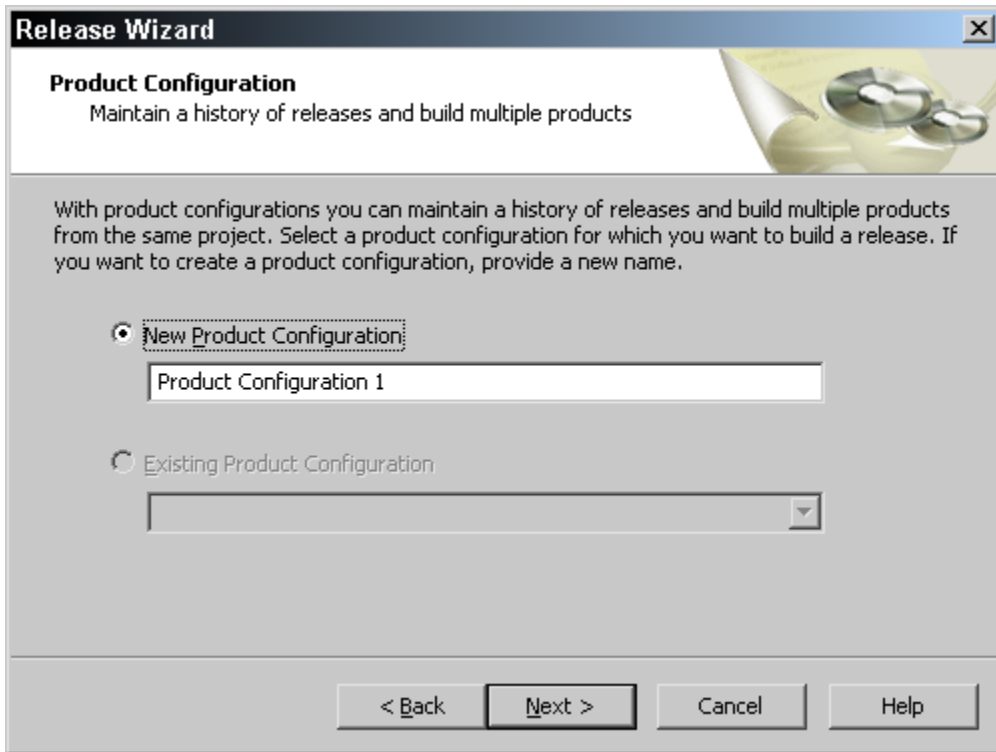


Figure 5-45: *The Product Configuration panel in the Release Wizard.*

The Product Configuration panel (Figure 5-45) provides a default name for the product configuration. For this project, there is no compelling reason to change it, so you can leave the default name. If this were not the first time that you were running the Release Wizard, you could select a product configuration that had already been created.

To select an already created configuration, select the Existing Product Configuration option and then select the product configuration that you want to rebuild from the drop-down menu. Click Next to go to the Specify a Release panel (Figure 5-46).

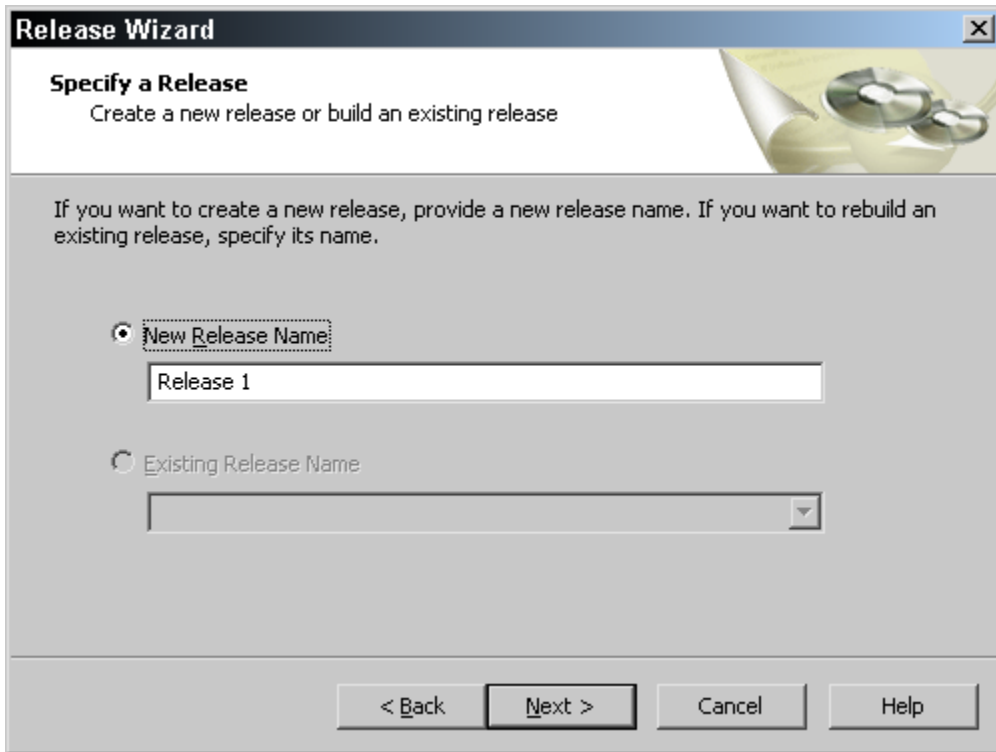


Figure 5-46: *The Specify a Release panel in the Release Wizard.*

When you specify a release name, it is added to the hierarchy that was started with the product configuration. There is a different set of properties that define a release than the set that defines a product configuration. Just as with product configuration, you can accept the default name. Click Next to move to the next panel.

The Filtering Settings panel allows you to filter features based on Release Flags that you have defined and to filter components based on the language of the component (Figure 5-47). As discussed earlier in this chapter, one of the properties that can be set for a feature is Release Flags. If you had identified a string as the release flag for a feature, you could then identify the release flag in the Release Flags field of the Filtering Settings panel. The build would then bring in only those features that had that particular release flag assigned or did not have any release flag specified (release flag independent).

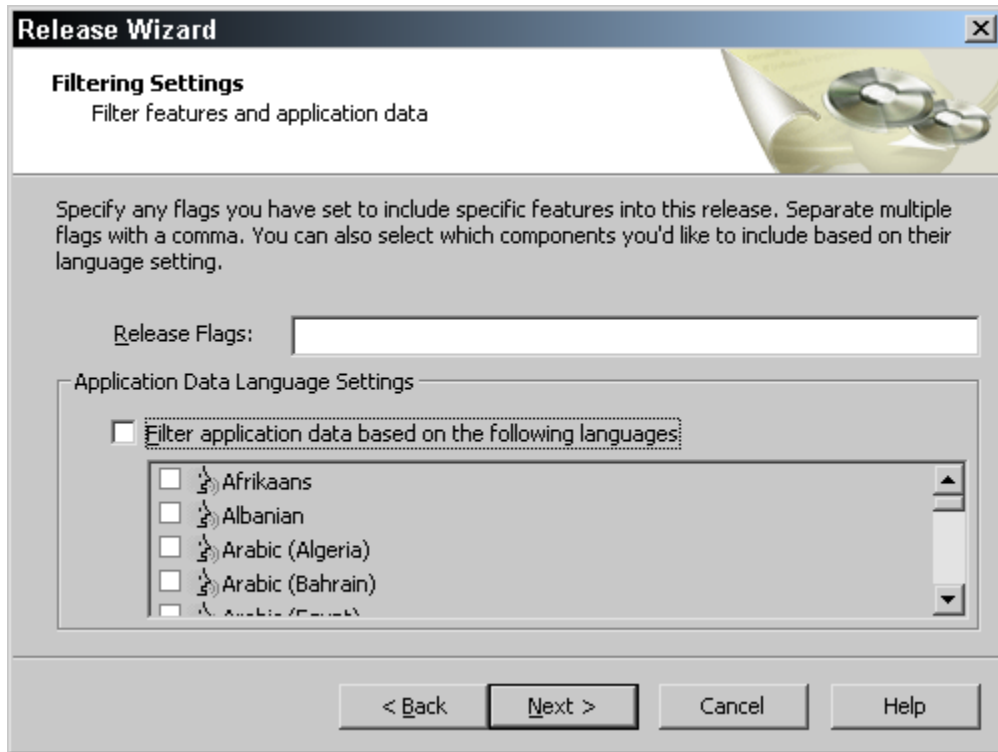


Figure 5-47: *The Filtering Settings panel in the Release Wizard.*

You can also filter application data based on a language or languages. This filtering is performed on the Languages property for components. This was discussed earlier in this chapter. If you select the check box just inside the group box below the Release Flags field, the list box of languages is enabled. You can then select the languages on which the filtering of components will be applied. This filtering will bring into the build those components that have defined the selected languages and will also bring in those components that are designated as language independent.

Being able to filter on features and on components provides a low level of what is called SKU (pronounced skew) management. The acronym SKU stands for Stock Keeping Unit and it refers to the ability to create various offshoots of a product. Filtering alone is not enough to create the various offshoots of a product and that is where the product configuration comes in. Of course with a small application like Developer Art, there are no concerns about filtering.

The next dialog in the Release Wizard is the Setup Languages panel (Figure 5-48). You use this dialog to create the functionality that allows the end user to select a language in which to run the installation user interface. You do not need to do anything on this panel because you are creating an English-language user interface for the Developer Art installation.



Figure 5-48: *The Setup Languages panel in the Release Wizard.*

The next dialog in the wizard is the Media Type panel. This panel is used to define the type and size of media that is to be used for distribution of the application (Figure 5-49). The Media Type drop-down menu offers a number of different media sizes. Selecting the correct media size is important since it determines how disk splitting is done. Disk splitting is where the distribution image is created in sizes that will fit on disks of the selected size. The default size is termed Network Image and this image is considered to be unlimited in size. This type of image is placed on a network drive so that it can be installed from that location to the desktop in a networked environment. For special media sizes there is a Custom option where you can define the size of the

disks that you want created. All of the examples in this book will use the default Network Image media type because the sample applications are so small.

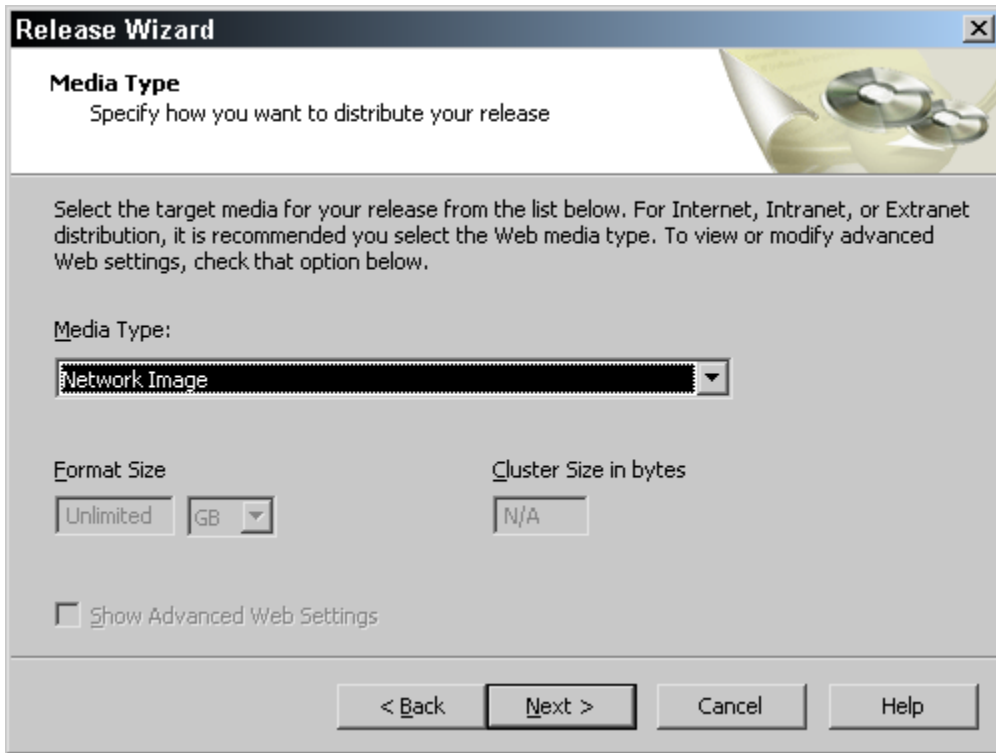


Figure 5-49: *The Media Type panel in the Release Wizard.*

After the Media Type panel comes the Release Configuration panel (Figure 5-50). In this panel, you can select to compress all the application files, leave the application files uncompressed, or have a mixture of compressed and uncompressed files. As discussed in Chapter 3, application files are compressed into Microsoft cabinet files that can either be embedded inside the MSI file or can be external to the MSI file.

For the Developer Art application, leave the default selection, which is to have all the application files uncompressed. If you were to choose to compress all the application files then you would get a single setup.exe file that contains all files including the MSI file as long as the media type selected in the previous dialog is a Network Image. For any other media types the cabinet files would be separate from setup.exe.

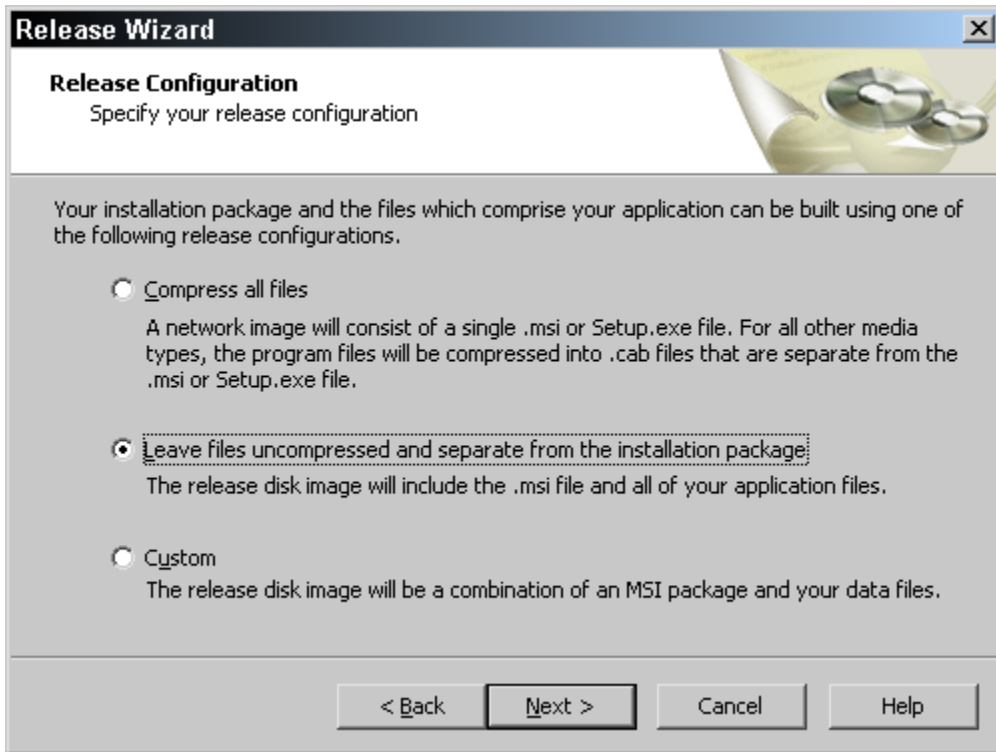


Figure 5-50: *The Release Configuration panel in the Release Wizard.*

The next dialog in the Release Wizard as we are using it to build the Developer Art installation program is the Setup Launcher panel (Figure 5-51). The purpose of this dialog is to allow you to control whether setup.exe is included as part of the installation package or not. This panel contains a disabled “Create installation launcher (Setup.exe)” check box. Since this is a Standard project it is required to have setup.exe as was discussed in Chapter 4, so this check box cannot be deselected.

In the drop-down combo box in the Setup Launcher dialog you can choose which of the Windows Installer engines you want included in you installation. You can select to have the Windows 9.x engine and the Windows NT/2000/XP engine, just one or the other of the engines, or neither Windows Installer engine. This option lets you choose the media image that is the smallest depending on the systems that you are going to target with your installation.

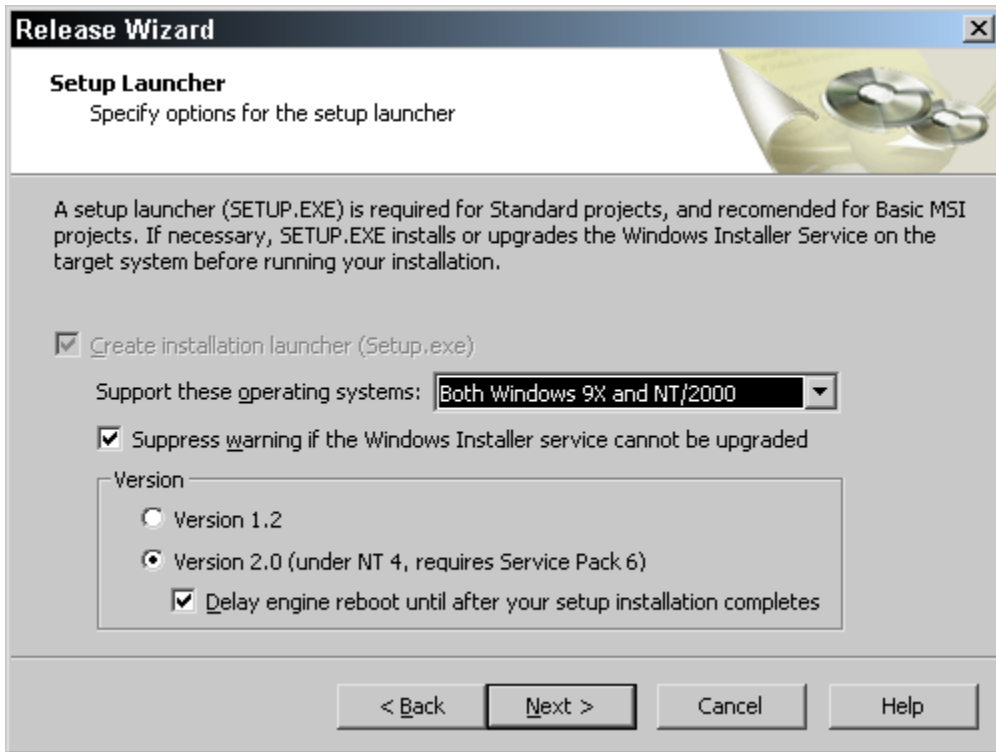


Figure 5-51: *The Setup Launcher panel in the Release Wizard.*

When setup.exe discovers that there is an earlier version of the Windows Installer engine on the target machine it will display a warning message box if it is not able to update the engine on the target system. This is a situation that occurs if you try to install version 1.2 of the Windows Installer engine on Windows 2000. Version 1.2 of the Windows Installer engine is not compatible with Windows 2000. If you do not want this warning message box displayed, you can suppress it by selecting the check box directly below the drop-down combo box. This is checked by default.

In the Version group box you can select to have either version 1.2 of the Windows Installer engine installed on the target system or to have version 2.0 installed. You should note that version 2.0 of the Windows Installer engine can only be installed on Windows NT 4.0 if Service Pack 6 has already been installed. When installing the Windows Installer engine, a reboot is required before the installation can be run. With version 2.0 of the Windows Installer engine it is possible to install it without the need for a reboot prior to running the installation. You can implement this reboot

functionality by making sure the check box at the bottom of the Version group box is selected. With this option selected the reboot is delayed until after the completion of the installation of the application. For the Developer Art application, you can leave all selections as their defaults.

The next dialog in the Release Wizard is the Windows Installer Location panel (Figure 5-52).

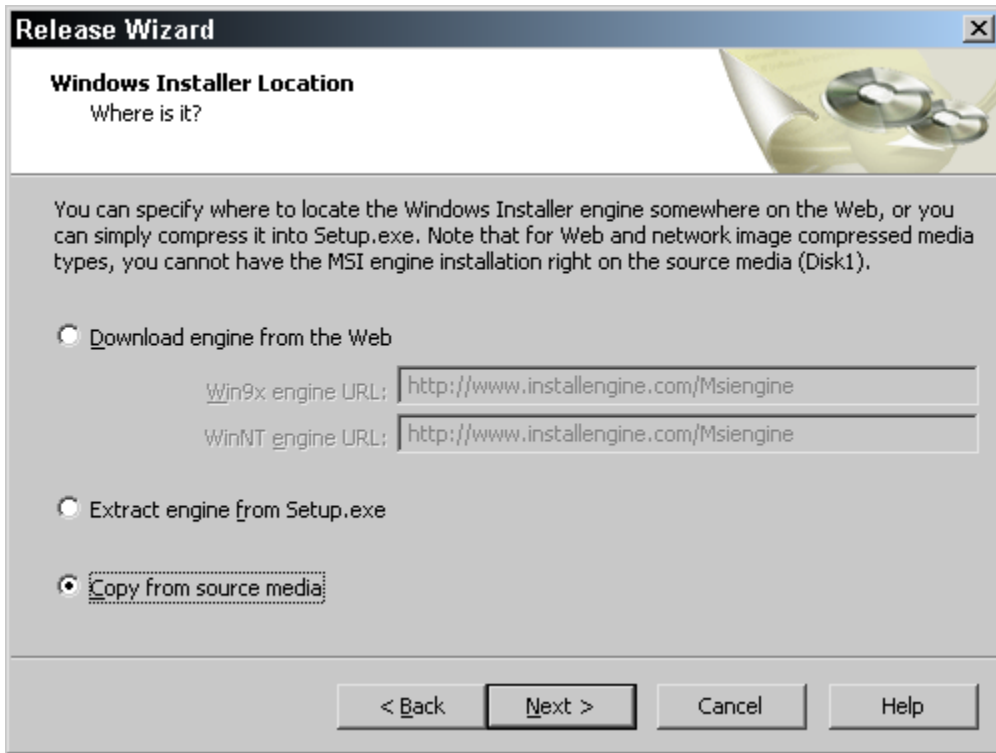


Figure 5-52: *The Windows Installer Location panel in the Release Wizard.*

This panel presents three choices for defining where the Windows Installer engine is to be found. The default location is for the engine to be directly on disk one of the distribution media and this is what is required for the Developer Art installation. There is also the “Extract engine from Setup.exe” option. If this option is selected, the Windows Installer engines will be compressed inside setup.exe during the build and the engine that needs to be installed will be extracted from setup.exe and then installed at run time. The top most option on this panel is selected if you do not want

to include the Windows Installer on the distribution media. The engine is hosted on the indicated Web site and the installation program can always be sure to install the latest engine if it is required.

The next dialog in the Release Wizard is the InstallScript Engine panel (Figure 5-53). This panel is very similar in purpose to the Windows Installer Location panel.

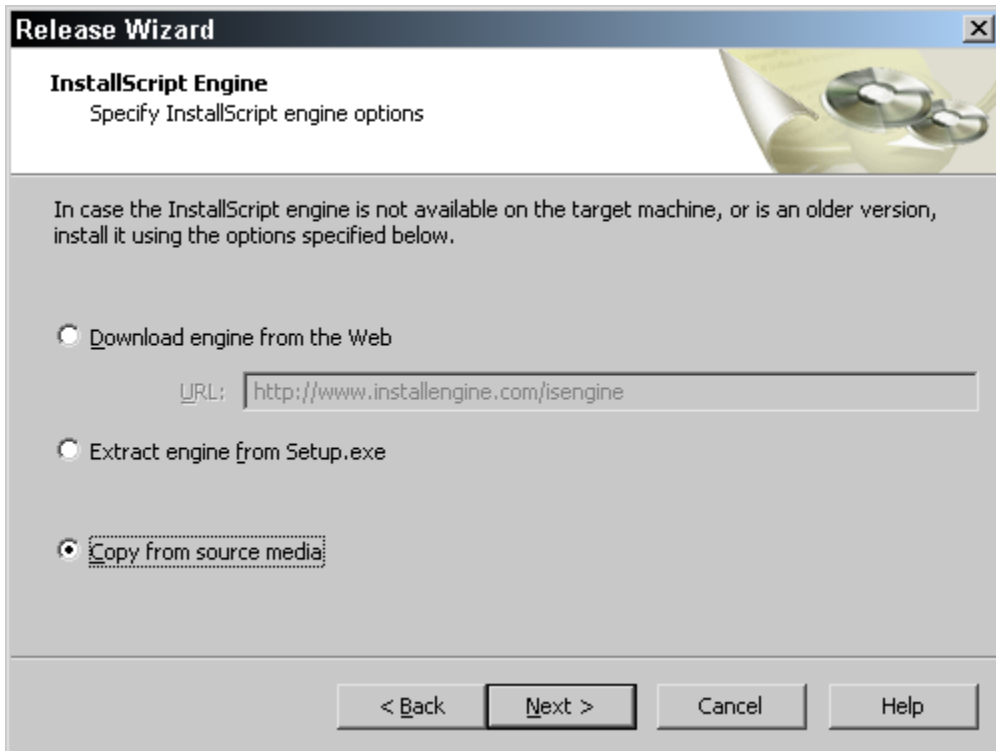


Figure 5-53: *The InstallScript Engine panel in the Release Wizard.*

In this dialog, you select the location from where the InstallScript engine is to be installed on the target system. The default is to include it on disk one of the distribution media and to install it from there. This is the selection you should use for the Developer Art project. The other two options on this dialog are to have it compressed inside setup.exe and extract it from that location before installing it or to install it directly from the indicated URL.

The dialog that follows the InstallScript Engine dialog, not shown, concerns the setting of .NET Framework options. Since the Developer Art application does not use .NET you want to leave the default selections in this dialog and move onto the next dialog.

After the .NET Framework dialog is the Advanced Settings panel (Figure 5-54). This panel allows you to specify some final settings before the build process is executed. At the top of this dialog is the Release Settings group box where you can select these settings. The Location field allows you to browse to a new location where you might want this particular build created. This overrides the standard location that is set in the File Locations tab of the Options dialog.

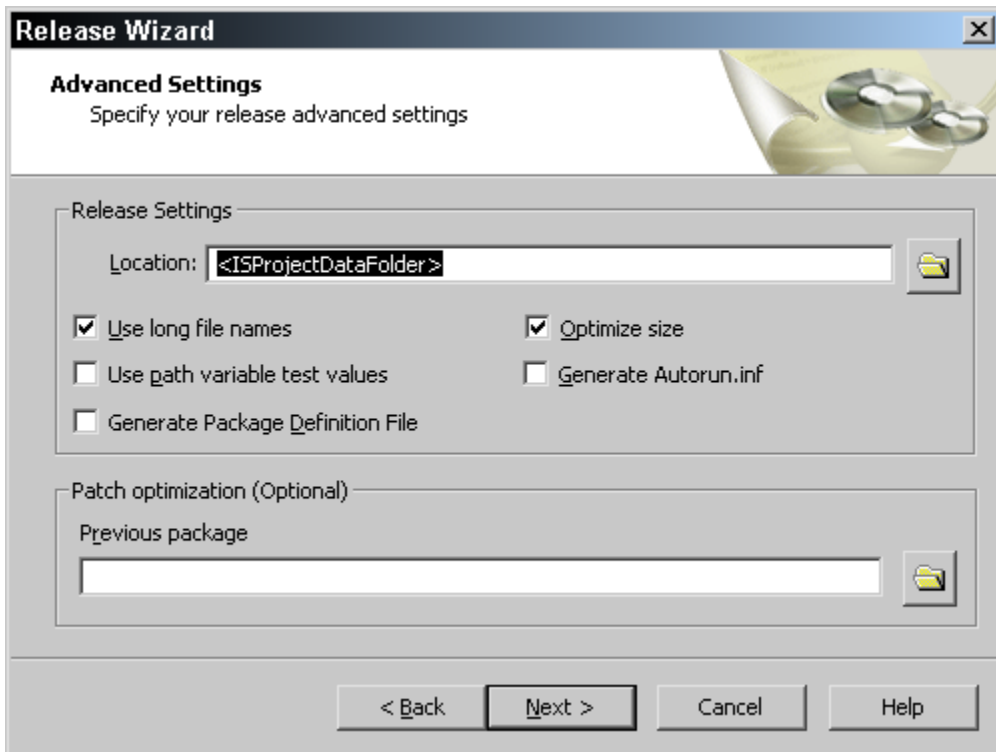


Figure 5-54: *The Advanced Settings panel in the Release Wizard.*

The default build location for this Standard project is defined by the path variable <ISProjectDataFolder> and it specifies the following path:

```
C:\MySetups\DeveloperArt_IDEStd\
```

You can click the button to right of the Location field and browse to a different location for this particular release. Making a change for a particular release does not affect the location that will be used for any other release.

Under the Location field there are five options that we need to discuss.

Use long file names: By default long file names are used to copy files to locations on the target machine. In Chapter 3 we learned that the Word Count property in the Summary Information Stream indicates to the Windows Installer engine whether files should be referred to with their long name or their short name. The only time that you would need to use short file names is when the target machine for some reason does not support long file names. When you deselect this option, the correct entry for the Word Count property is made in the Summary Information Stream.

Use path variable test values: This is where the test value for a path variable is used in place of the normal location at which a path variable points. This is a mechanism that is useful when there are two locations where the source files for an application can be found. For example you may have your normal location for your source files on a network server and a backup location on the hard drive of the build machine. If for some reason the network connection is down and you still need to make a build, you can select this option and obtain the source files from the local hard drive.

Generate Package Definition File: Selecting this option creates a PDF file that is used for deploying applications using SMS. This file is created in the root of the installation image.

Optimize size: With this option selected, the compression algorithm that will be used is the LZJ compression supported by the MAKECAB.EXE utility. This provides a higher compression but the build time is longer. If files are not being compressed, selecting this option has no effect.

Generate Autorun.inf: When you select this option, an AUTORUN.INF file will be created at the root of the installation image. This file allows for the automatic running of an installation on a CD-ROM as soon as the CD is inserted into the drive.

At the bottom of the Advanced Settings dialog there is another group box labeled Patch Optimization (Optional). This is important if you are performing a build that will be used to create an upgrade patch. Patching is not discussed in this book.

The final dialog in the Release Wizard is the Summary panel (Figure 5-55). This panel allows you to review all the selections that were made throughout the wizard. If you need to make any changes, you can go back and make them. When you are satisfied with the selections, click Build to launch the build process.

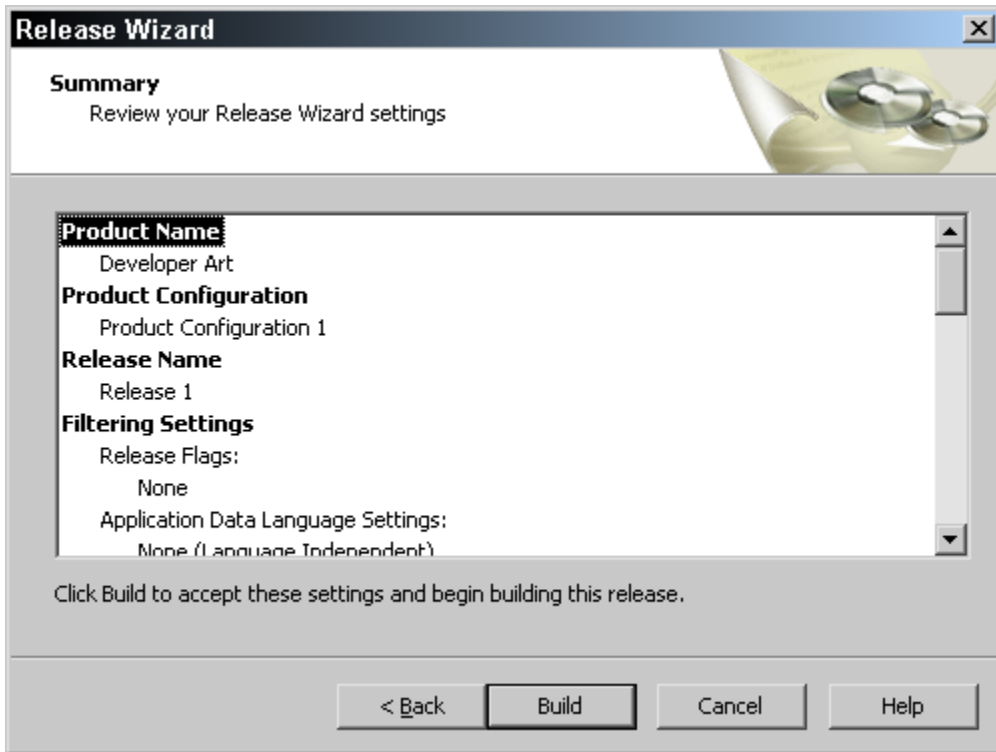


Figure 5-55: *The Summary panel in the Release Wizard.*

The Developer Art project build should complete with zero errors and zero warnings. The progress of the build process is displayed at the bottom of the screen in the Build output window. When the build is finished and you select the Releases view, you should see something like what is shown in Figure 5-56.

If you click on the Product Configuration 1 sub-view in the sub-view list you will see a list of the properties that define a product configuration. If you click on the Release 1 sub-view you will see another longer set of properties that go into defining a release of a particular product configuration.

If you expand the tree under the Release 1 sub-view you will see that we have access to a text log file that captures the build output that was displayed in the output window. There is also a report that provides a summary of the make up of the installation program that was built. This summary includes names of the files in the build and the number of features in the application.

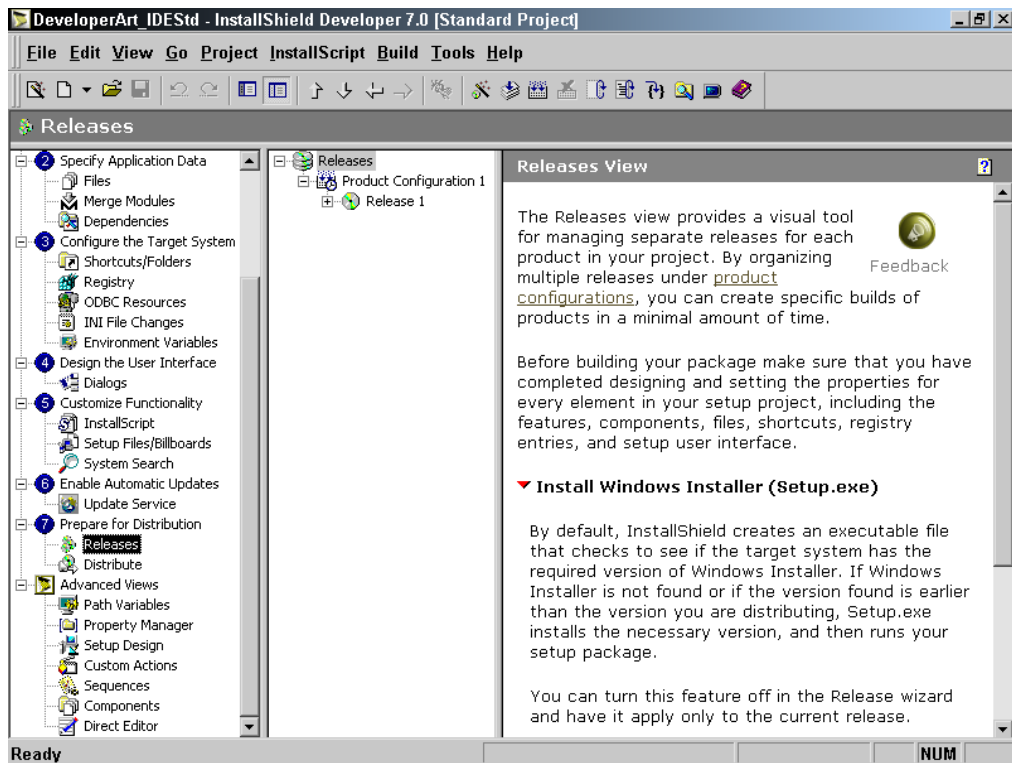


Figure 5-56: *The Releases view after the completion of the build.*

This has been a brief introduction to the Release Wizard. There are many more features in this wizard that are not within the scope of this book.

The final proof of any build is actually installing the application and seeing that it works. Do this by clicking the Run button on the toolbar. You will see that the sequence of dialogs in the user interface is different than it was for the installation created by the Project Wizard.

You are now ready to do the exercise for the creation of a Basic MSI installation package in the IDE. The discussion that we had relative to the creation of the Standard project applies almost in total to the creation of the Basic MSI installation project.

Creating a Basic MSI Project in the IDE

Creating a Basic MSI installation package for the Developer Art is left as an exercise. It is recommended that the approach used to create this project is to use the Advanced Views to define the project structure and create the shortcut. What you have done in the first part of this chapter should provide plenty of information about what you need to do to create this project.

It is suggested that the name of the project be `DeveloperArt_IDEMSI.ism` to distinguish it from the Standard project. When creating this project, note the differences in the property sets for features and components, as well as the slight differences in the Release Wizard between the building of the Standard installation program and the Basic MSI installation package.

When working in the Setup Design view, you should take the opportunity to provide component names that are not the names of the files in the component. When you are working in the Setup Design or the Components view and you are creating the component that is to install `ArtWork.dll`, remember to set the Extract at Build property to Yes. A completed Basic MSI project is available on the included CD-ROM at the back of the book.

As with the Standard project, you should run the installation and make sure that the application runs correctly. You should also watch the user interface carefully to note any differences between the Standard project approach and the Basic MSI approach.

Conclusion

In this chapter, you have seen a direct relationship between the InstallShield Developer IDE and the installation program that is being authored. In addition, you learned that working in the IDE provides a lot more capability than is available in the Project Wizard.

Creating a Standard project and creating a Basic MSI project in the IDE are very similar operations. There are only a few differences in the properties that need to be set for features and components. These additional properties for a Standard project make it very easy to upgrade from earlier projects created by InstallShield Professional – Standard Edition.

InstallShield Developer has the robust capability to create many different installation packages from one project file. This ability enables what is called SKU management (product configuration) making for an efficient control of the various flavors of a product.

You will use the projects that you created in this chapter in the chapters of Part III. You will be adding functionality to these particular projects as you work through the examples that are provided.

Part II

The InstallScript Language

Variables and Data Types

InstallScript is a complete programming language that facilitates installation program creation. A program created with InstallScript appears similar to a program created using the C language. InstallScript is not the C language though it does have to be compiled and linked before it can be run. The fact that it has to be compiled makes it much like the Java language, which is compiled into byte code that is then interpreted by a virtual machine. InstallScript is compiled, but it requires the InstallScript engine for the compiled script to be executed.

Chapter 4 discusses the InstallShield Developer run-time architecture. This and subsequent chapters in Part II discuss InstallScript as a programming language. The only way to learn a programming language is to use it, so let's begin.

Setting up the Programming Environment

To learn a programming language, you need to be able to write code and you need a feedback mechanism to verify that your programs are working correctly. In this section, you will create a project whose only purpose is to run InstallScript programs.

First, launch InstallShield Developer. In the InstallShield Today view, go to the Create a new project sub-view and create a new Standard project under `C:\MySetups` called "Learning InstallScript". To configure this project:

1. Go to Project Properties under the General Information view and make entries for the Setup Author Name and the Authoring Comments properties.
2. In the Summary Information Stream sub-view, make entries for the Subject and Comments properties. Skip the Add/Remove Programs sub-view since this project will not install an application.
3. In the Product Properties view, make an entry for the Product Name property. It is not necessary to modify any of the other properties for this sub-view since this project will not install any files.
4. In the Sequences view, under Advanced Views, go to the Installation folder. Expand the tree for the User Interface table and remove all the actions except the ISVerifyScriptingRuntime custom action. To remove the actions, right click on each action and select Remove. The result should be as shown in Figure 6-1.
5. Click the Build button on the toolbar to create a build using default selections. There will be one warning that there are no files included in the build. You can ignore this warning.

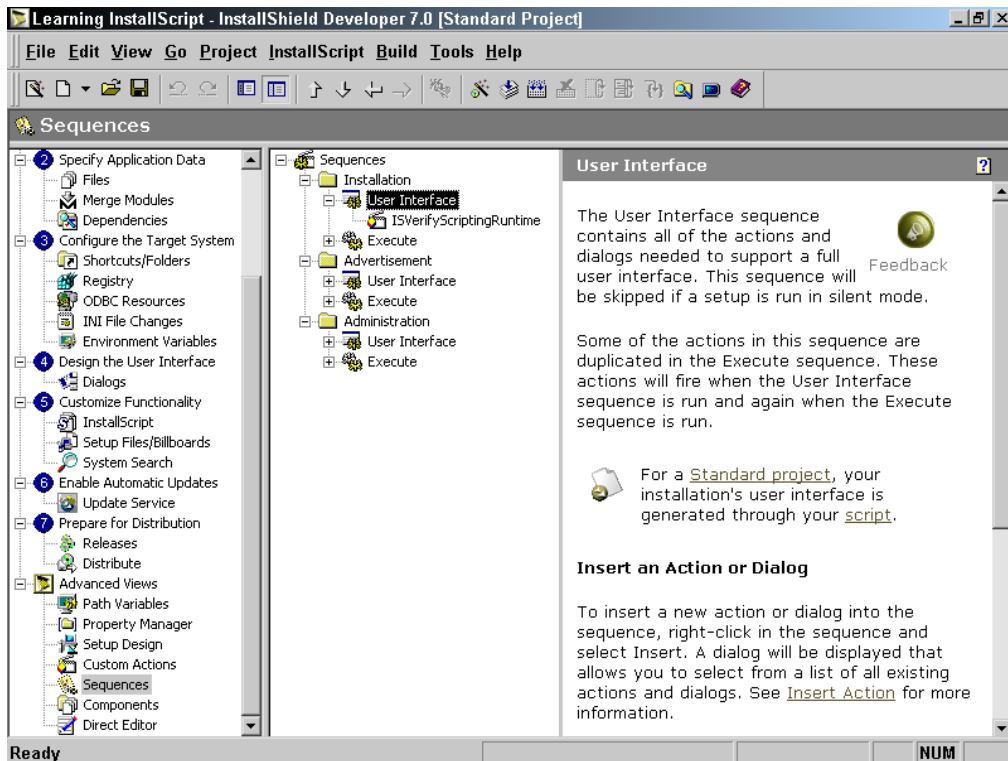


Figure 6-1: *The User Interface sequence view after removing all but one of the actions.*

When learning to program with a new language, you need an efficient means to display the output from sample programs. InstallShield Developer provides a Windows message box that can be used to display program output. While practicing with InstallScript, you do not want to contend with any type of user interface other than the message box, which displays your program output.

To implement the basic scripting framework to learn the InstallScript language, go to the InstallScript view and click on the file Setup.Rul. This displays the Script Editor. Replace the default code with the code that is shown in Figure 6-2. All InstallScript code shown in this book as figures can be found on the CD-ROM at the back of the book. The code for each figure is in a separate .rul file and name of the file is the name of the figure.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script is for learning how to
//                 program the InstallScript language as
//                 discussed in the book Getting Started with
//                 InstallShield Developer and Windows
//                 Installer Setups.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION    "Feedback"

program

    SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
               "Learning InstallScript");

endprogram

```

Figure 6-2: *Setup.rul* to be used as the starting point for learning *InstallScript*.

InstallScript provides two message box-style functions that could display feedback for sample programs. These are the `SprintfBox` and the `MessageBox` functions. This exercise uses the `SprintfBox` function because it makes it easier to display the values of numbers and other data types. The `SprintfBox` function is a wrapper around the `wsprintf` C library function and the `MessageBox` Windows API. Since this project has already been built, you do not have to rebuild it. Every time that you change InstallScript code, all you have to do is compile it and the new compiled script will be streamed into the Binary table of the database. After you have typed in the code shown in Figure 6-2, compile it. If there are no warnings or errors, run the installation package by clicking the Run button on the toolbar. A message box should appear with the string "Learning InstallScript."

When you run the sample program, an initialization dialog appears prior to the message box. To simplify the learning environment, you can eliminate this initialization dialog so that all you see is the message box. Do this by making a change in the `Setup.ini` file that is part of the media image and is located at the root of this image. The location of this file under `C:\MySetups` is shown in Figure 6-3.

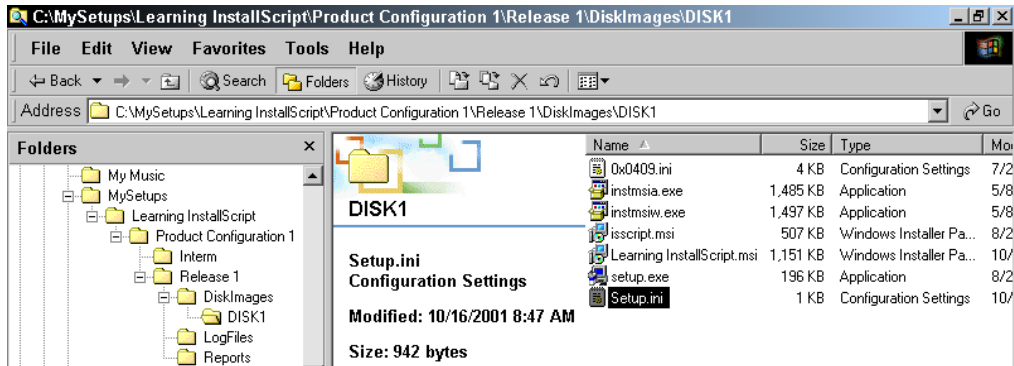


Figure 6-3: The root location for the *Learning InstallScript* media image.

Open the Setup.ini file and add another line to the [Startup] section. As shown below, add the line UI=0 at the end of the section. This is a supported option that you can use to help you learn how to program with InstallScript. Note that you should not build the project again because that would delete this file and replace it with the default Setup.ini. If you compile your script only, then you do not have to modify the Setup.ini file again.

```
[Startup]
CmdLine=
SuppressWrongOS=Y
ScriptDriven=1
ScriptVer=4.0.0.110
Product=Learning InstallScript
PackageName=Learning InstallScript.msi
MsiVersion=1.20.1827.0
EnableLangDlg=N
DoMaintenance=Y
UI=0
```

In Chapter 4 there is a complete discussion of all the entries in the Setup.ini file. After making this change in Setup.ini, rerun the project by clicking the Run button on the toolbar. Verify that the initialization dialog does not appear.

Before you move on, you can make some changes to the Script Editor functionality to make it easier to use. In the Script Editor, right click and select Properties. In the Window Properties dialog, make the following changes:

- **Color/Font tab:** Select Left Margin from the Items list box. From the Color drop-down menu, select a color to display the left margin
- **Misc tab:** Make sure that the “Confine caret to text” option is selected. This forces the caret in the Script Editor to fall back to the end of the present line if you click to the right of the end of the line. This prevents you from inadvertently typing where you do not want to type.

Variables

A variable is a symbolic name that represents a particular location in the computer's memory. In InstallScript, a variable name has to be declared before it can be used, unlike what can be done in a language like VBScript. The general form of a variable declaration is as follows:

```
DataType VariableList;
```

In this general form `DataType` needs to be a valid data type as defined by the InstallScript language and `VariableList` is one or more valid variable names. Commas separate multiple variable names and the line must end with a semi-colon. The `DataType` tells the compiler how to interpret the information at the memory location to which the variable name points.

A variable name can be constructed from uppercase and lowercase letters, digits, and the underscore character (`_`). The first character of a variable name must be a letter or the underscore character. Digits cannot be used as the first character of a variable name. A variable name can be any length, but only the first 62 characters are significant. If there are two variable names each made up of one hundred characters, the first 62 characters of each variable name has to be unique. If not, the compiler assumes that you are declaring the same variable twice and it generates a compiler error.

Variable names are case sensitive and creating good variable names is one way to create self-documenting code. Variable names should have mnemonic significance so the variable name indicates both the purpose and the data type of the variable. Common among Windows programmers is the Hungarian notation method of variable naming. This system became widely used inside Microsoft. It came to be

known as Hungarian notation because the prefixes make the variable names look a bit as though they are written in a language other than English and because the inventor, Dr. Charles Simonyi, is originally from Hungary.

In Hungarian Notation, a variable name begins with one or more lowercase letters that indicate the variable's data type, followed by a name made up of mixed-case letters that indicate the variable's use. In InstallScript the use of Hungarian notation is limited to identifying the data types that are supported by the language. You can create your own conventions to be used when you write InstallScript code. Table 6-1 shows one such convention that can be used.

Table 6-1: Hungarian Notation Convention for InstallScript

Prefix	Data Type	Example
sz	STRING	szLastName – a zero terminated string that represents the last name of a person.
b	BOOL	bFound - a Boolean that shows the success of a search.
c	CHAR	cLower – a character that holds a lower case letter.
h	HWND	hDlgItem – a handle to a control on a dialog box.
i	INT	iIndex – an integer being used as an index.
l	LONG	lCount – a long integer being used to hold the value of a counting operation.
p	POINTER	pArray – a pointer to an array.
obj	OBJECT	objFSO – a FileSystemObject created by using the CreateObject function in InstallScript.

The examples in Table 6-1 give you an idea of how you can create your own convention for naming variables. The important thing about any convention that you

create is that it needs to make sense to someone else that reads your code. The InstallScript Help Library uses Hungarian Notation for the descriptions of built-in functions and examples. A complete discussion of this notation style can be found in many introductory Windows programming books. Also the MSDN Library contains a reprint of the original monograph written by Dr. Charles Simonyi describing the Hungarian notation identifier naming convention.

You can declare variables in three different locations: inside functions, in the definition of function parameters, and outside all functions. In InstallScript, the code between the program and the endprogram keywords does not define a function even though it operates as a main function. Therefore, you cannot define variables inside this code block.

Variables that are defined inside of a function are local variables, variables that are defined in a function definition are the formal parameters to the function, and variables that are declared outside all functions are global variables. Local variables and formal parameters are accessible only inside the function where they are declared and global variables are accessible from anywhere in the program. Chapter 8 discusses functions in more detail.

As an experiment, declare two string variables and two integer variables, assign them values, and then display them in a message box. The InstallScript code for doing this is shown in Figure 6-4.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script is for learning how to
//                program the InstallScript language as
//                discussed in the book Getting Started with
//                InstallShield Developer and Windows
//                Installer Setups.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION   "Feedback"

```

Figure 6-4: *Setup.rul for the variable declaration exercise.*

```

STRING  szStr1, szSTR2;
INT     iNum1, iNUM2;

program
    szStr1 = "String1";
    szSTR2 = "String2";
    iNum1 = 100;
    iNUM2 = 1000;

    SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
               "szStr1 = %s\nszSTR2 = %s\niNum1 = %d\niNUM2 = %d",
               szStr1, szSTR2, iNum1, iNUM2);

endprogram

```

Figure 6-4: *Continued.*

After you type this code in the Script Editor, compile it, but do not build it. Then run the installation program to see the output (Figure 6-5). Note that the code sample in Figure 6-4 declared the variables above the `program` keyword, but set their values between the `program` and the `endprogram` keywords. Study the input to the `SprintfBox` function to make sure that you understand it. The first argument to this function is a combination of Windows MessageBox styles that defines that the message box should have one OK button and that the icon displayed in the message box should be the information icon, a lower case `i`.

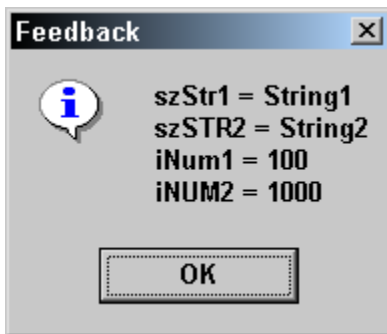


Figure 6-5: *The output message box for the variable declaration exercise.*

The second argument to this function is the string constant that you declared before the `program` block that is displayed in the message box's title bar. The third argument is a formatting string that is used to display the output in the message box.

The `%s` specification prints out the corresponding argument as a string and the `%d` specification prints out the corresponding argument as an integer. The last arguments are the variables for which you want to print their values. Each argument here has an associated type specification that defines how the value of the argument is to be displayed. There is a `%s` specification for each of the string variables and a `%d` specification for each of the integer variables. The variable arguments in the input to the `SprintfBox` function are passed in the same order as the output specifications in the format string.

Data Types

In InstallScript, as in most programming languages, there are built-in data types and user-defined data types.

The Built-In Data Types

In InstallScript there are four basic data types. One of these data types has a number of aliases. InstallScript supports only numbers, strings, variants, and objects. These four basic data types are defined below and then discussed in detail in separate subsections. In this book, the convention for declaring a variable as a particular data type is to specify the data type using all uppercase letters.

NUMBER: In InstallScript, numbers are implemented as a four-byte signed integer. This means that, within InstallScript, a variable of type NUMBER can represent values from $-2,147,483,648$ to $2,147,483,647$. All variables that are declared as type NUMBER are initialized to 0.

STRING: A variable of type STRING is an array of characters. When InstallShield Developer is hosted on Windows NT or Windows 2000, a STRING variable is handled as a Unicode array of characters, which means that each character is two bytes in size. When installed on Windows 9.x machines, strings are handled as multi-byte characters. Any variable declared as type STRING is initialized to a null string ("").

Variables of the STRING data type on Windows NT and Windows 2000 are handled as a BSTR. A BSTR is a length-prefixed string. The length is stored as an

integer at the memory location preceding the data in the string. A BSTR is null terminated after the last counted character, but may also contain null characters embedded within the string. The string length is determined by the character count, not the first null character.

VARIANT: Visual Basic and VBScript programmers are familiar with this data type. VBScript has only one data type called a Variant. As in VBScript, a VARIANT is a special kind of data type in InstallScript that can contain different kinds of information, depending on how it is used. At its simplest, a VARIANT can contain either numeric or string information. A VARIANT behaves as a number when it is used in a numeric context and as a string when used in a string context. That is, if you are working with data that looks like numbers, InstallScript assumes that it is numbers and does what is most appropriate for numbers. Similarly, if the data can only be string data, InstallScript treats it as string data. Variables declared as type VARIANT are initialized as empty.

OBJECT: This data type is an IDispatch interface pointer and is used to access COM objects from InstallScript. (Chapter 9 discusses COM and InstallScript.) Variables declared as type OBJECT are initialized as empty.

There are no floating-point data types, which means that InstallScript supports only integer math.

The NUMBER Data Type

There are a number of aliases for the NUMBER data type. It is important to not use the NUMBER data type when declaring a variable but to use the aliases that describe how the program uses the variable. Currently, all the aliases for the NUMBER data type mean that the variable is a four-byte signed integer, but this may not be the case in the future. For example, a CHAR, which now is still a four-byte integer, might mean one byte in a future InstallScript language enhancement.

The aliases for the NUMBER data type are listed below:

BOOL: Used to represent the TRUE or FALSE condition.

CHAR: Used to represent a single character. Since a single character can be represented by one byte it is the lower byte of the four-byte number that is used to represent the character. There is one situation where a CHAR data type is only

one byte in length and that is when this data type is used in a structure. Structures are discussed in the section on user-defined data types.

HWND: Used to declare variables that are used to hold handles to windows or any handle returned by the Windows operating system.

INT: Used to declare a variable that will hold a four-byte signed integer.

LIST: Used to declare a pointer to an InstallScript internal implementation of a linked list.

LONG: Used to declare a variable that will hold a four-byte signed integer.

LPSTR: Used to declare a variable that will hold a pointer to a null-terminated string.

POINTER: Used to declare a variable that will hold a generic pointer.

PSZ: Used to declare a variable that will hold a pointer to a null-terminated string.

SHORT: Used to declare a variable that will hold a four-byte signed integer. There is one situation where a SHORT data type is only two bytes in length and that is when this data type is used in a structure. Structures are discussed in the section on user-defined data types.

The following sections provide sample scripts to help you understand how to use the data types.

BOOL

A variable that is declared as a BOOL data type has a value of TRUE or FALSE. When a BOOL data type is FALSE it has a value of 0. When it has a value of TRUE it can have any value other than 0. The following program shows the usage of this data type. The program calculates the greatest common divisor between two integers.

In this program, the variable `bFinished` is used to control the looping done in the `while` loop. This variable is initially set to FALSE. As soon as the variable `iSmall` becomes zero, the `bFinished` variable is set to TRUE and the looping stops. After

the looping stops, the original values and their greatest common divisor are displayed in the feedback message box.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This program calculates the greatest
//                common divisor between two integers.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION    "Feedback"

BOOL    bFinished;
INT     iLarge, iSmall, iRemainder;
INT     iLargeOrig, iSmallOrig;

program

    bFinished = FALSE; // Initialize BOOL data type to FALSE
    iLarge = 1517;
    iSmall = 369;
    iLargeOrig = iLarge; // Save the original value
    iSmallOrig = iSmall; // Save the original value

    // Loop until bFinished is set to TRUE
    while(!bFinished)
        iRemainder = iLarge % iSmall;
        iLarge = iSmall;
        iSmall = iRemainder;

        // Check to see if looping should be stopped
        if(iSmall = 0)then
            bFinished = TRUE;
        endif;
    endwhile;

    // Print out the input and the GCD
    SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
               "The greatest common divisor of %d and %d is %d",
               iLargeOrig, iSmallOrig, iLarge);

endprogram

```

Figure 6-6: *Setup.rul* for calculating the greatest common divisor of two integers.

CHAR

Although a variable declared as type CHAR can represent any number that can be represented by a four-byte signed integer, this type of variable should be used only to represent individual characters. A string is an array of characters, but it is not the same as an array of type CHAR. Since a CHAR in InstallScript is a signed four-byte integer, an array of type CHAR in InstallScript is an array of integers. The following program shows the use of the CHAR data type.

```

////////////////////////////////////
//
//  File Name:      Setup.rul
//
//  Description:   Learning the InstallScript language
//
//  Comments:     This program displays the value of
//                an ANSI character code and the character.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION   "Feedback"

CHAR    cANSIChar;

program

    cANSIChar = 190; // Set the CHAR variable to a displayable
                    // ANSI character code.

    // Print out both the ANSI character code
    // and the character that it represents.
    SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
               "The ANSI code %d represents the %c character",
               cANSIChar, cANSIChar);

    cANSIChar = 'Z'; // Set the CHAR variable to a character value

    // Print out both the character and the
    // ANSI character code that it represents.
    SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
               "The character %c has the ANSI character code %d",
               cANSIChar, cANSIChar);

endprogram

```

Figure 6-7: *Setup.rul* for displaying character values and ANSI character codes.

This program first sets a variable of type CHAR to one of the displayable ANSI character codes and then prints out the variable as a character and as an integer. The second part of the program reverses this operation by setting the variable to an ANSI character and printing out the ANSI code and the character.

HWND

Declaring a variable of the HWND type indicates that it is being used as the handle to a window. It is also currently used for other handles because there are no aliases that are more specific to the other handle types. The following program captures the window handle for InstallShield Developer and displays it in hexadecimal format.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This program displays the value of the
//                window handle for InstallShield Developer.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION   "Feedback"

HWND    hwndISDev;

program

    // Get the window handle to InstallShield Developer
    // using a built-in function from InstallScript.
    hwndISDev = FindWindow("InstallShieldIDE",
        "Learning InstallScript - InstallShield Developer 7.0" +
        " [Standard Project]");

    // Print out the hex value for the Window handle
    // for InstallShield Developer.
    sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
        "The InstallShield Developer window handle is %#.8x",
        hwndISDev);

endprogram

```

Figure 6-8: *Setup.rul* for obtaining the handle to the InstallShield Developer window.

The above program uses `FindWindow`, which is one of the built-in InstallScript functions. For the first argument, this function requires the name of the class to which the InstallShield Developer main window belongs or a null string. The second argument is the title of the window for which the function is trying to get the handle. This title is displayed in the title bar of the main InstallShield Developer window. If you have access to Visual Studio, you can use the Spy++ tool to obtain the class name for the main InstallShield Developer window. You can also get the value of the window handle to verify what the above program returns as its value.

Note the format string used to print out the value of the window handle. This format takes into account that the window handle is usually displayed has a hexadecimal number and that this number is a four-byte unsigned number. The format string used identifies that the output value is in hexadecimal form and the number is eight digits wide with zeros used to pad the value to the left.

INTEGER

The integer data types consist of the `INT`, `LONG`, and `SHORT` aliases. In a normal program these types can all be used interchangeably. Inside of a structure, however, an `INT` and a `LONG` are the same four-byte signed integers, but a `SHORT` is a two-byte signed integer. In a 32-bit operating system environment, there is no difference between an integer and a long integer so in InstallScript you can use the `INT` and `LONG` data types interchangeably. Note that you should use the `SHORT` data type only where you want to specify a two-byte integer.

In InstallScript, the fact that integers are signed is important only in the context of the script. When integer values are passed to functions they are implicitly type cast into the integer type that the receiving function requires. In the program that is listed in Figure 6-9, variables are defined as type `LONG` and these variables are assigned values at the very size limit of what a four-byte value can hold. This program then displays these values as signed integers and as unsigned integers.

As this program demonstrates, even if InstallScript treats a number as a four-byte signed integer, when it is passed to a function that expects an unsigned integer, the function uses it as an unsigned integer. The output from this program is shown in Figure 6-10.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This program prints out the value of the
//                various large integers both as signed
//                and unsigned values.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION    "Feedback"

LONG    iLarge1, iLarge2, iLarge3;

program

    // Create two large integer values
    iLarge1 = 2147483648;
    iLarge2 = 2147483647;

    // Add the two large values together
    iLarge3 = iLarge1 + iLarge2;

    // Display the large values as both signed and
    // unsigned values using the sprintfBox function.
    sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
        "iLarge1 as a signed integer: %d\n\n" +
        "iLarge1 as an unsigned integer: %u\n\n" +
        "iLarge2 as a signed integer: %d\n\n" +
        "iLarge2 as an unsigned integer: %u\n\n" +
        "iLarge3 as a signed integer: %d\n\n" +
        "iLarge3 as an unsigned integer: %u",
            iLarge1, iLarge1, iLarge2, iLarge2,
            iLarge3, iLarge3);

endprogram

```

Figure 6-9: *Setup.rul* for showing the handling of large integer values in *InstallScript*.

To completely understand the results of this program (Figure 6-10) you might want to independently investigate how a computer handles negative numbers. This involves the designation of the highest order bit as the sign bit and the use of twos complement to enable the subtraction of two numbers.

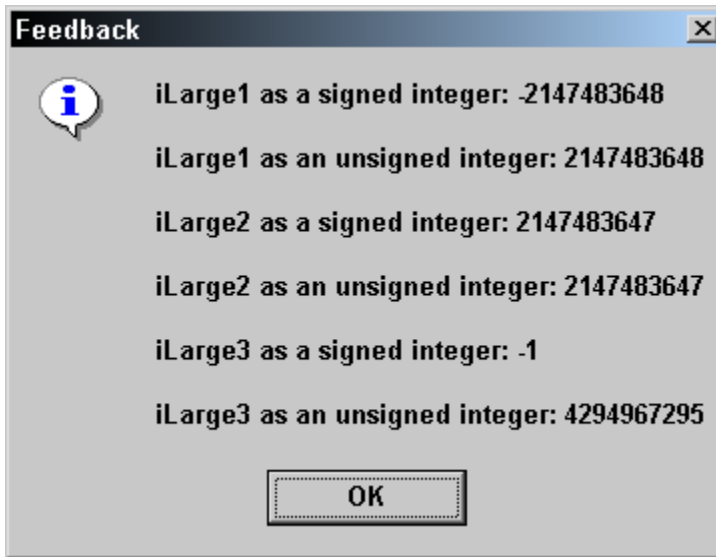


Figure 6-10: *The output from the program in Figure 6-9.*

InstallScript's use of signed integers is a concern only when your code contains math.

POINTER

There are three explicit pointer data types in InstallScript: PSZ, LPSTR, and POINTER. Though the LIST data type is also a pointer, it is a special case that is covered separately. In the following program, Figure 6-11, you create and manipulate pointers to strings and integers.

This program shows the use of the address of operator (&) and the dereference operator (*). It shows that a pointer to a string can be incremented to point to parts of the string. Pointers are also used to manipulate variables of the INT data type. The dereference operator is used to obtain the value stored at the memory address defined by a POINTER variable. The dereference operator can be used only with variables that have the basic NUMBER data type. You can have pointers to pointers, but to dereference a pointer, you have to do it one level of indirection at a time. The value of pointers is for passing these values to functions exported by DLLs. Pointers are not used inside InstallScript except in special situations.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This program displays the value of strings
//                and integers that are defined through the use
//                of pointers.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION    "Feedback"

STRING    sString;
INT       iValue1, iValue2, iValue3;
LPSTR     pStr;
POINTER   pNum1, pNum2, pNum3, ppNum;

program

    sString = "abcdefghi";

    // Get a pointer to the string
    pStr = &sString;

    // Print out different parts of the string by
    // incrementing the string pointer
    SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
        "%9s\n\n%9s\n\n%9s\n\n%9s\n\n%9s\n\n%9s\n\n%9s\n\n%9s\n\n%9s",
            pStr, pStr+1, pStr+2, pStr+3, pStr+4,
            pStr+5, pStr+6, pStr+7, pStr+8);

    iValue1 = 100;

    // Set the value of the pointer using the
    // the address of operator &
    pNum1 = &iValue1;

    // Using the dereference operator * to
    // add 100 to the value of iValue1
    iValue2 = *pNum1 + 100;

    // Set the value of the pointer using the
    // the address of operator &
    pNum2 = &iValue2;

```

Figure 6-11: *Setup.rul* that demonstrates the handling of the pointer data types.

```

// Get a pointer to the pointer to iValue2
ppNum = &pNum2;

// Dereference the pointer to the pointer to iValue2
// We cannot double dereference a pointer to a pointer
// in one statement, has to be done one level at a time.
pNum3 = *ppNum;

// Set the value of iValue3 by adding 100
// to the value of iValue2.
iValue3 = *pNum3 + 100;

// Print out the value of the integer data types
// and the value of the pointer data types.
sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
    "iValue1 = %d\n\npNum1 = %p\n\niValue2 = %d\n\n" +
    "pNum2 = %p\n\nppNum = %p\n\npNum3 = %p\n\niValue3 = %d",
    iValue1, pNum1, iValue2, pNum2, ppNum, pNum3, iValue3);

endprogram

```

Figure 6-11: *Continued.*

LIST

A LIST is a doubly linked list that can hold numerical values or string values. A variable that is declared as type LIST is actually a pointer to a doubly linked list. InstallScript has implemented many functions to allow you to easily work with the LIST data type. The LIST data type and the STRING data type are the only data types that have a set of functions specifically designed to work with variables declared as these types.

There are three basic types of operations that you can perform on a variable of type LIST: creation, querying, and destruction of the list; adding, modifying, and removing values; and traversing the list. To sort a list or perform a binary search of a list, you have to create your own InstallScript functions to perform these types of sophisticated operations.

Below is an example, Figure 6-12, of using a LIST data type to reverse the digits in a number. This example demonstrates some of the functions that used to manipulate the LIST data type.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This program uses a string list to reverse
//                the digits in a number and then displays
//                the revised number.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION    "Feedback"

LIST    lstStr; // Define a LIST variable
INT     iType, iLastDigit, iNum, iReturn;
STRING  svBuf, szNum;

program

    // Create a pointer to a string list
    lstStr = ListCreate(STRINGLIST);

    // Define the number to be reversed.
    iNum = 1234567890;

    // Starting with the last digit extract the digit,
    // convert it into a string, and add it as an
    // element to the string list.
    while(iNum != 0)
        iLastDigit = iNum % 10;
        Sprintf(svBuf, "%d", iLastDigit);
        ListAddString(lstStr, svBuf, AFTER);
        iNum = iNum/10;
    endwhile;

    // Set the pointer to the first element in the list
    iReturn = ListSetIndex(lstStr, LISTFIRST);

    // Loop through the list, extract the value from
    // the element, and concatenate it to the output variable.
    while(iReturn != END_OF_LIST)
        ListCurrentString(lstStr, svBuf);
        szNum = szNum + svBuf;
        iReturn = ListSetIndex(lstStr, LISTNEXT);
    endwhile;

```

Figure 6-12: *Setup.rul* for demonstrating the use of the LIST data type.


```

// Destroy the list now that it is no longer needed.
ListDestroy(lstStr);

// Display the reversed number as a string
PrintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
          "Reversed number = %s", szNum);

endprogram

```

Figure 6-12: *Continued.*

This program could have done something similar using a `NUMBERLIST` instead of a `STRINGLIST`, but the above example demonstrates the basic approach that is needed to work with the `LIST` data type. To find the description of all the built-in functions that are available for working with the `LIST` data type, see the InstallScript Language Reference. The Language Reference contains a List Processing Functions topic that lists the list-related functions and links to their descriptions. To access the Language Reference, place your cursor in the Script Editor and press F1.

The first thing that the above program did was declare a variable of the `LIST` data type and then use the `ListCreate` function to create a list that would hold strings. It then extracted each digit from the back of the number, turned it into a string using the `Sprintf` built-in function and then placed it into the list. Finally the program traversed the list to extract the string elements to build the number in reversed form. Before displaying the results, the program destroyed the list using the `ListDestroy` built-in function. You should always do this to release memory that is no longer required.

The **STRING** Data Type

There are no aliases for the `STRING` data type so a string variable is always declared as type `STRING`. InstallScript provides a number of functions to manipulate `STRING` variables. You can find descriptions of these functions in the InstallScript Language Reference. The string function descriptions are found in the String Functions topic. The first example program using strings demonstrates a method for reversing the characters in a string (Figure 6-13).

The technique used in this example swaps the characters in the string with using a temporary variable. This technique uses the exclusive OR bitwise operator. This operator is discussed in Chapter 7.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This program reverses the letters in a string.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION    "Feedback"

STRING  szStr;
INT i, j, iLen;

program

    szStr = "This is a string";

    // Get the length of the string to be reversed.
    iLen = StrLength(szStr);

    i = 0; // Set i to point to first string
    j = iLen - 1; // Set j to point at last character

    // Work from both ends of the string to
    // swap the characters without using a
    // temporary variable to perform the swap.
    while(i < j)
        szStr[i] = szStr[i] ^ szStr[j];
        szStr[j] = szStr[j] ^ szStr[i];
        szStr[i] = szStr[i] ^ szStr[j];

        i++; // Increment i index
        j--; // Decrement j index
    endwhile;

    // Print the value of the reversed string.
    sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
               "Reversed string = %s", szStr);

endprogram

```

Figure 6-13: *Setup.rul demonstrating reversing letters in a string.*

This program creates a STRING variable without specifying a size. When an initial size is not specified, the string automatically takes on the size required for the string

that is assigned as its value. Strings can be declared with a minimum size by putting the size of the string in square brackets. To specify a variable of type `STRING` with a minimum size, use the following format:

```
STRING    szString[20];
```

This declaration creates a variable that will have a minimum size of 20 bytes. If you want to assign a string that is longer than 20 bytes, this is not a problem because strings in InstallScript are auto-sized. Unless there is a good reason to set a minimum size for a string variable, it is best to allow the InstallScript auto-sizing to size the string to whatever is necessary.

After giving the variable a value, the program uses the `StrLength` function to get the length of the string in bytes. It then loops through half the length of the string swapping the characters at each end with each other. The approach used for swapping the characters uses a technique that does not require the use of a temporary variable. Note that there is an increment and decrement operator used to adjust the values of the two indices. The postfix version of the increment and decrement operators is supported but the prefix version is not supported.

The next example shows one approach to performing a case-sensitive comparison of two strings (Figure 6-14). The function `StrCompare`, one of the built-in InstallScript functions, does a comparison that is not case sensitive.

```

////////////////////////////////////
//
//  File Name:      Setup.rul
//
//  Description:   Learning the InstallScript language
//
//  Comments:     This program performs a case sensitive
//                comparison of two strings.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION   "Feedback"

STRING  szStr1, szStr2, szResult;
INT     iLen1, iLen2, i;

```

Figure 6-14: *Setup.rul demonstrates the case sensitive comparison between two strings.*

```

program
    i = 0;
    szStr1 = "THIS IS A STRING";
    szStr2 = "This is a string";

    // Get the length of each string
    iLen1 = StrLengthChars(szStr1);
    iLen2 = StrLengthChars(szStr2);

    // Check for the first character that is different
    // between the two strings. Since an uppercase letter
    // has a lower ANSI value a string of all uppercase
    // letters will be considered to be less than the same
    // string that is all lowercase letters.
    while(szStr1[i] != '\0' && szStr2[i] != '\0')
        if(szStr1[i] < szStr2[i]) then
            szResult = "String one is less than string two";
            goto DisplayResult; // Jump to label to print result
        elseif(szStr1[i] > szStr2[i]) then
            szResult = "String one is greater than string two";
            goto DisplayResult; // Jump to label to print result
        endif;
        i++;
    endwhile;

    // If all characters are the same up to the end of the
    // shortest string then the length of the two strings
    // determines the which string is greater than the other
    // whether they are equal.
    if(iLen1 = iLen2) then
        szResult = "String one and string two are equal";
    elseif(iLen1 < iLen2) then
        szResult = "String one is less than string two";
    elseif(iLen1 > iLen2) then
        szResult = "String one is greater than string two";
    endif;

DisplayResult:
    // Print the results of the string comparison.
    sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION, "%s", szResult);

endprogram

```

Figure 6-14: *Continued.*

Figures 6-13 and 6-14 show some of the different ways that you can work with strings. Mainly, these programs show that you can look at the individual characters that make up a string. The example in Figure 6-14 used the `StrLengthChars` function

instead of the `StrLength` function. In this example, the result is the same because it does not use a multi-byte character string.

The VARIANT Data Type

The `VARIANT` data type can hold a string value or an integer value. A variable of type `VARIANT` can be used to convert a number to a string or a string to a number. This is demonstrated in the code listed in Figure 6-15.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This program demonstrates the usage
//                of the VARIANT data type.
//
////////////////////////////////////
#include "ifx.h"

#define CAPTION   "Feedback"

VARIANT vValue1, vValue2;
INT      iValue1, iValue2;
STRING  szValue1, szValue2;

program

    vValue1 = "12345"; // A VARIANT set as a string
    vValue2 = 67890;  // A VARIANT set as a string
    iValue1 = vValue1; // Convert string to an integer
    szValue1 = vValue2; // Convert integer to a string

    iValue1 = iValue1 + 67890; // Addition
    szValue1 = "12345" + szValue1; // Concatenation
    iValue2 = vValue1 + 67890; // Addition
    szValue2 = "12345" + vValue2; // Concatenation

    // Print the results of the data manipulation.
    SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
               "iValue1 = %d\n\nszValue1 = %s" +
               "iValue2 = %d\n\nszValue2 = %s",
               iValue1, szValue1, iValue2, szValue2);

endprogram

```

Figure 6-15: *Setup.rul* showing the use of the `VARIANT` data type.

The above program shows the use of a VARIANT data type to convert between string and integer data types. It also shows that a variable of type VARIANT in an expression will be treated appropriately according to the context in which it is being used. This technique for converting between strings and numbers using a VARIANT data type is used in a number of the examples shown in this book. Using a VARIANT in this fashion does away with the need for calling the StrToNum and NumToStr InstallScript functions. However, there is probably a performance penalty for using a VARIANT data type in this fashion.

The OBJECT Data Type

A variable of type OBJECT in InstallScript is used to access automation objects that are available on the target system. We will discuss COM in more detail in Chapter 9, but the program in Figure 6-16 shows a small application using this data type.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This program demonstrates the usage
//                of the OBJECT data type.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION    "Feedback"

OBJECT  objWI, objStrList;
INT     iCompCnt;
program

    try
        // Create a Windows installer object.
        set objWI = CreateObject("WindowsInstaller.Installer");

        // Create a string list object that holds all the
        // components on the build system.
        set objStrList = objWI.Components;

```

Figure 6-16: *Setup.rul demonstrating the use of the OBJECT data type.*

```

        // Capture the number of components on the build system.
        iCompCnt = objStrList.Count;

        // Print the number of components.
        SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
            "The number of components on this machine is %d", iCompCnt);

    catch
        SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
            "Exception thrown when creating a Windows Installer object");
    endcatch;

    // Free the memory taken by the objects.
    set objStrList = NOTHING;
    set objWI = NOTHING;

endprogram

```

Figure 6-16: *Continued.*

It is not important to completely understand this program at this time. This program uses the automation interface for the Windows Installer to query the build system and get a value of how many Windows Installer components are installed.

User-Defined Data Types

In InstallScript you can define two data types that do not fall under the built-in data type definition. These two data types are arrays and structures.

The Array

The array data type allows you to create a collection of a specific built-in data type. The array can be considered a possible replacement of the LIST built-in data type. Working with an array does not require a set of functions to manipulate it. All you need is the index, which provides fast random access that a LIST does not provide. Just as with the STRING data type, an array can be declared with a specific size or it can be declared with a zero size. An array is declared of a particular built-in data type as shown in the following example.

```

INT    iArray1(), iArray2(10);

```

This example declares one array of integers that has an initial size of zero and another array of integers that has an initial size of ten. Just like strings, an array has zero-based indexing, (that is, the first element in an array has an index of 0). Unlike strings, an array is not automatically sized to accommodate the number of values we want it to hold. An array has to be specifically sized for the number of elements it is going to hold. For this reason InstallScript has two special operators for this purpose: `SizeOf` and `Resize`.

The `SizeOf` operator returns the present size of an array and the `Resize` operator allows you to change the size of an array at run time. However, if an array is initially declared as a specific size, it cannot be resized to a smaller size than originally declared. For the most efficient use of memory you should declare all arrays as having zero size and then resize them according to the need of the run-time environment.

The following program shows the declaration and use of an array to hold the first twenty Fibonacci numbers (Figure 6-17). Fibonacci numbers is a sequence of numbers where except for the first two numbers in the sequence each number is equal to the sum of the previous two numbers. The first two numbers are defined to be 0 and 1 respectively. Fibonacci numbers have many applications in the field of mathematics and in the study of computer algorithms. They are also one of the standard ways to show the manipulation of arrays.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This program demonstrates how to use the
//                 array data type to store the values of
//                 the first 20 Fibonacci numbers.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION    "Feedback"

INT      iFibonacci(); // Integer array of zero size
INT      i, iNumFibonacci;
STRING   szDisplay;
VARIANT  vFibonacci;

```

Figure 6-17: *Setup.rul* for demonstrating the array data type.


```

program

    iNumFibonacci = 20;

    // Size the array to what is required.
    Resize(iFibonacci, iNumFibonacci);

    // Set the first two Fibonacci numbers
    iFibonacci(0) = 0;
    iFibonacci(1) = 1;

    // Use a VARIANT to convert a number to a string
    // and use this string to create an output display.
    vFibonacci = iFibonacci(0);
    szDisplay = vFibonacci;
    vFibonacci = iFibonacci(1);
    szDisplay = szDisplay + ", " + vFibonacci;

    // Calculate the remaining members of the
    // array of Fibonacci numbers.
    for i=2 to iNumFibonacci-1
        iFibonacci(i) = iFibonacci(i-1) + iFibonacci(i-2);

        // Add each new number to the output string.
        vFibonacci = iFibonacci(i);
        szDisplay = szDisplay + ", " + vFibonacci;
    endfor;

    // Print the Fibonacci numbers in the array.
    SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
        "The first %d Fibonacci numbers: \n%s",
            iNumFibonacci, szDisplay);
endprogram

```

Figure 6-17: *Continued.*

This program initially declares an integer array of zero size by using a set of empty parentheses. It then resizes this array to that specified by the variable `iNumFibonacci`. As is required with the calculation of Fibonacci numbers, the first two elements in the array are defined. In order to display all the numbers that are calculated, the program creates a `STRING` variable and uses a `VARIANT` to convert the elements in the array to a string. As each of the Fibonacci numbers is calculated in the loop, the program adds each new number to this display string. Finally the program displays the values of the Fibonacci numbers in a single message box.

The Structure Data Type

While the array data type allows you to create a collection of values of the same data type, a structure allows you to create a collection of different data types. This is a very powerful mechanism for collecting one-unit variables that are logically related but not all of the same data type. A good example of logically related variables that are not of the same data type is the values that make up a row in one of the database tables in an .msi file.

The program in Figure 6-18 shows the declaration of a structure to hold a person's name and birthday. The declaration needs to explicitly declare any string members with a specific size. In the structure, a variable of type CHAR is only one byte and a variable of type SHORT is only two bytes. This is demonstrated when the program takes the size of this structure. To define a structure you need to use the typedef statement and then declare a variable of this type. To access the members of the structure, use the structure member reference operator, which is a period (.).

The structure defined in this program defines three members of type STRING (two of which have 25 bytes and one that has 10 bytes), one member of type CHAR, and two members of type SHORT. Inside a structure, a CHAR member takes up one byte and a SHORT member takes up two bytes. Therefore, any variable declared of type BIRTHDAY will take up 65 bytes in memory. Using the sizeof operator and displaying its return value confirms this. Note the use of the VARIANT type variable to convert numbers to strings to make the date display easier.

```

////////////////////////////////////
//
//   File Name:      Setup.rul
//
//   Description:   Learning the InstallScript language
//
//   Comments:     This program demonstrates how to use the
//                 structure data type to store a person's
//                 birth date.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION    "Feedback"

```

Figure 6-18: *Setup.rul demonstrating the use of the structure data type.*

```

// Define a structure that holds
// information on a person's birthday.
typedef BIRTHDAY
begin
    STRING    szLastName[25];
    STRING    szFirstName[25];
    CHAR      cMiddleInitial;
    STRING    szMonth[10];
    SHORT     iDay;
    SHORT     iYear;
end;

// Declare a variable of type BIRTHDAY.
BIRTHDAY    MyBirthday;
VARIANT     vDay, vYear;
STRING      szDate;
INT         iSize;

program

    // Assign values to the members of MyBirthday
    // using the structure member reference operator.
    MyBirthday.szLastName = "Baker";
    MyBirthday.szFirstName = "Robert";
    MyBirthday.cMiddleInitial = 'S';
    MyBirthday.szMonth = "August";
    MyBirthday.iDay = 4;
    MyBirthday.iYear = 1939;
    // Create a display for the birthday date.
    vDay = MyBirthday.iDay;
    vYear = MyBirthday.iYear;
    szDate = MyBirthday.szMonth + " " + vDay + ", " + vYear;

    // Get size of the structure.
    iSize = SizeOf(MyBirthday);

    // Print the structure members and structure size.
    SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
        "Name: %s %c. %s\n\nDate: %s\n\nSize of structure: %d",
        MyBirthday.szFirstName, MyBirthday.cMiddleInitial,
        MyBirthday.szLastName, szDate, iSize);

endprogram

```

Figure 6-18: *Continued.*

There is another means to access the members of a structure. This approach is demonstrated in the following program (Figure 6-19). This program uses the

structure pointer operator instead of the structure member reference operator. The program demonstrates that you can use a structure as a member of another structure. Here a structure holds a person's name and another structure holds a date. Combining these two structures in another structure is used to create the BIRTHDAY structure.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This program demonstrates how to use a
//                pointer to a structure data type to store
a person's birthdate.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION    "Feedback"

// Define a structure to hold a date.
typedef BIRTHDATE
begin
    STRING    szMonth[10];
    SHORT     iDay;
    SHORT     iYear;
end;

// Define a structure to hold a name.
typedef NAME
begin
    STRING    szLastName[25];
    STRING    szFirstName[25];
    CHAR      cMiddleInitial;
end;

// Define a structure that holds
// information on a person's birthday.
typedef BIRTHDAY
begin
    NAME      MyName;
    BIRTHDATE MyBirthDate;
end;

```

Figure 6-19: *Setup.rul* that demonstrates structure pointers.

```

// Declare a variable of type BIRTHDAY.
BIRTHDAY    MyBirthday;
BIRTHDAY POINTER    pMyBirthday;
VARIANT     vDay, vYear;
STRING      szDate;
INT         iSize;

program

    // Get pointer to the MyBirthday structure.
    pMyBirthday = &MyBirthday;

    // Assign values to the members of MyBirthday
    // using the dot operator.
    pMyBirthday->MyName.szLastName = "Baker";
    pMyBirthday->MyName.szFirstName = "Robert";
    pMyBirthday->MyName.cMiddleInitial = 'S';
    pMyBirthday->MyBirthDate.szMonth = "August";
    pMyBirthday->MyBirthDate.iDay = 4;
    pMyBirthday->MyBirthDate.iYear = 1939;

    // Create a display for the birthday date.
    vDay = pMyBirthday->MyBirthDate.iDay;
    vYear = pMyBirthday->MyBirthDate.iYear;
    szDate = pMyBirthday->MyBirthDate.szMonth + " " +
              vDay + ", " + vYear;

    // Get size of the structure.
    iSize = SizeOf(MyBirthday);
    // Display the structure members and structure size.
    sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
        "Name: %s %c. %s\n\nDate: %s\n\nSize of structure: %d",
        pMyBirthday->MyName.szFirstName,
        pMyBirthday->MyName.cMiddleInitial,
        pMyBirthday->MyName.szLastName, szDate, iSize);

endprogram

```

Figure 6-19: *Continued.*

The program shown in Figure 6-19 shows two important techniques. These are the fact that a structure can have other structures as its members and that we can use pointers to structures to access the members of a structure. The structure pointer operator is the combination of the dash and the right angle bracket without any spaces between (->). You declare a pointer to a structure by using the typedef name of the structure and the generic POINTER data type name as shown in Figure 6-19. You assign a value to this pointer by using the address of operator (&).

Conclusion

Most of this chapter discussed the built-in data types and the user-defined data types. There are four built-in data types: NUMBER, STRING, VARIANT, and OBJECT. The NUMBER data type has several aliases that you should use instead of declaring all number variables as NUMBER. The other three data types have no aliases. The discussions of these data types demonstrates that the VARIANT data type can be used to convert from number to string or from string to number without having to call one of the built-in InstallScript functions.

The end of the chapter demonstrated the use arrays to create collections of values of the same built-in data type. It also showed how to use a structure to create a collection of related data even if this data consisted of variables with different data types. This discussion covered how to access the members of a structure using either the structure member reference operator (.) or the structure member reference pointer operator (->). You also saw that a structure could have other structures as members.

Expressions and Statements

In InstallScript, you declare variables using the data types discussed in Chapter 6 and combine them to create expressions. Chapter 6 provided a number of code examples that demonstrated the use of the data types. As required, these code examples used expressions. This chapter goes further to discuss the various operators that are used to create expressions. An expression is a valid combination of operators, variables, and constants.

A statement is any part of code that can be executed. The statements covered in this chapter are those built-in statements that allow you to control the flow of a program's execution. These statements can be divided into three categories, statements that are used to select something, statements that are used to repeatedly execute the same lines of code, and statements that allow you to jump to somewhere else in the code.

Expressions

The discussion of expressions revolves around the operators that are available for each of the data types discussed in the last chapter. Operators can be broken down into five categories:

Arithmetic: In InstallScript, an expression of this type is used to manipulate integer numbers. Remember that InstallScript does not support floating-point math.

Relational and Logical: Relational and logical expressions are those that evaluate to either TRUE or FALSE.

String: A string expression is one that is primarily related to concatenation, searching for a sub-string, or an expression that accesses a string table entry.

Bitwise: Bitwise expressions are focused on testing, setting, or shifting the bits in a variable.

SizeOf/Resize: Expressions that use these two operators relate to manipulating arrays.

Arithmetic Expressions

In InstallScript there are seven operators that can be used in arithmetic expressions.

Table 7-1: Arithmetic Operators

Operator	Action	Description
*	Multiplication	$x * y$ multiplies the values of x and y .
/	Division	x/y divides the value of x by y .

Table 7-1: Arithmetic Operators (Continued)

Operator	Action	Description
%	Modulus	$x \% y$ returns the remainder of the division of x by y .
+	Addition	$x + y$ returns the sum of x and y .
-	Subtraction and unary minus	$x - y$ returns the difference between x and y . $-x$ negates the value of x .
++	Increment	$x++$ increments the value of x by 1.
--	Decrement	$x--$ decrements the value of x by 1.

Because InstallScript deals with signed integers, a mathematical operation that produces a value larger than 2 GB means that the sign of the result changes to negative. This happens because, with signed integers, the highest order bit is the sign bit. When it is set, the number is evaluated as negative.

Figure 7-1 provides the code for a simple program that displays the results of various arithmetic expressions. This program does not declare enough variables to hold the results of each expression. Instead the expressions are used as the arguments to the `PrintfBox` function. In this example, the expression is evaluated as it is passed to the function. The expressions used in this program show the results for all the arithmetic operators except the increment and the decrement operators.

The increment and decrement operators cannot be used in a normal expression. They can be used in postfix position to increment or decrement a variable where the variable and the increment or decrement operator form the complete expression. Refer to Figure 6-13 in Chapter 6 to see how increment and decrement operators are used. In this example, the operators are used to traverse through the characters of a string in order to reverse the characters in the string. This is the only way to use the increment and decrement operators.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:       This script is demonstrates the results
//                  of various arithmetic expressions.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION    "Feedback"

INT a, b, c, d, e;

program

    // Assign values to the variables.
    a = 100;
    b = 2;
    c = 25;
    d = 3;
    e = 2000000000;

    // Print out the results of various arithmetic expressions.
    // Save space by placing the actual expression in the
    // sprintfBox function as the argument to be printed.
    sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
        "b - a = %d\n"          +
        "b * c = %d\n"          +
        "-b * c = %d\n"         +
        "a/c = %d\n"           +
        "a %% c = %d\n"         +
        "b/d = %d\n"           +
        "b %% d = %d\n"         +
        "a + b * c = %d\n"      +
        "a * b + c * d = %d\n"  +
        "a/b + c/d = %d\n"      +
        "b * e = %d",
        b-a, b*c, -b*c, a/c, a%c, b/d, b%d, a+b*c,
        a*b+c*d, a/b+c/d, b*e);

endprogram

```

Figure 7-1: *Setup.rul* that demonstrates the results from various arithmetic expressions.

The program in Figure 7-1 shows what happens when the variables *a*, *b*, *c*, *d*, and *e* are combined in various ways. The output of the program in Figure 7-1 is shown in Figure 7-2.

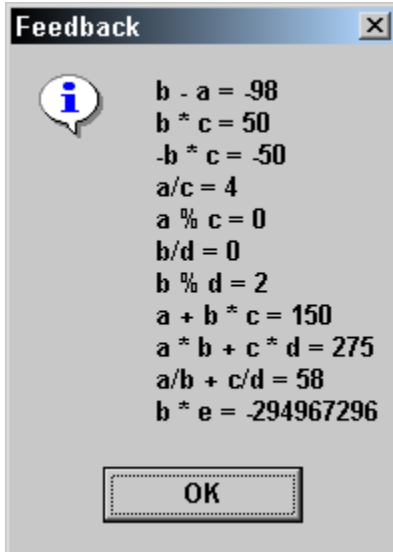


Figure 7-2: *The output from the program in Figure 7-1.*

The expressions that deserve additional attention are discussed below:

-b * c: This expression demonstrates the impact of using the minus sign in a unary fashion. In this expression, the minus sign reverses the sign of the variable *b* before the multiplication takes place. The unary minus operator has a higher precedence than multiplication.

a % c: This expression returns the remainder of the division between the variable *a* and the variable *c*. The fact that the result returned from this expression is 0 means that the value of *c* divided evenly into the value of *a*. It is a common technique to set *c* to 2 and evaluate whether *a* is even or odd.

b/d: This expression shows that integer division truncates the result to the smallest whole value. In this case the result is zero because the value of the variable *d* is larger than the value of the variable *b*.

a * b + c * d: This expression shows that there is a precedence to arithmetic operations. The result of this expression shows that multiplication is performed first, and then addition. Multiplication has a higher precedence than addition. Multiplication and division have higher precedence than addition and subtraction.

b * e: This expression demonstrates the impact of creating a number larger than 2 GB. Because InstallScript uses signed integers, you should ensure that results of all expressions are within the 2 GB boundary.

The increment and decrement operators can be used only as shown below:

```
i++; // This is a valid expression.
j--; // This is also a valid expression.
```

The postfix form of the increment and decrement operators shown above are the only valid forms. The prefix version of these operators is not valid. The only valid use of the increment and decrement operators in InstallScript is in the looping through a series of expressions and incrementing or decrementing an index during this looping.

When creating arithmetic expressions you should be aware of the precedence of the various operators. A summary of the precedence of the arithmetic operators is shown below from highest to lowest.

Highest	-(unary minus)
	*, /, %
Lowest	+, -

Since the increment and decrement operators in InstallScript cannot be used with any other operators, this chapter does not discuss their precedence. When you are not sure about how the precedence rules will be used to evaluate a certain expression, you can use parentheses to force the required precedence. Parentheses have the highest precedence of all operators. For example, if you want to change the normal precedence of the following expression so the addition happens before multiplication, you could use parentheses as shown below:

```
x = a * b + c * d; // Multiplication first, then addition
x = a * (b + c) * d; // Addition before multiplication
```

Relational and Logical Expressions

Because relational and logical operators work together this section covers both. A relational operator evaluates the relationship between two expressions. A relational operator is a binary operator because it requires two expressions. Logical operators are used to connect relational expressions together.

The result of a relational expression is either TRUE or FALSE. As in the C language, FALSE is zero and TRUE is non-zero. In the evaluation of a relational expression, if the result is TRUE then the result has a value of 1. Table 7-2 describes the relational operators supported by InstallScript. Table 7-3 describes the supported logical operators.

Table 7-2: The Relational Operators

Operator	Action	Description
>	Greater than	$x > y$ returns TRUE if expression x is greater than expression y ; otherwise, it returns FALSE.
>=	Greater than or equal	$x >= y$ returns TRUE if expression x is equal to or greater than expression y ; otherwise, it returns FALSE.
<	Less than	$x < y$ returns TRUE if expression x is less than expression y ; otherwise, it returns FALSE.
<=	Less than or equal	$x <= y$ returns TRUE if expression x is equal to or less than expression y ; otherwise, it returns FALSE.
=	Equal	$x = y$ returns TRUE if expression x is equal to expression y ; otherwise, it returns FALSE.

Table 7-2: The Relational Operators (Continued)

Operator	Action	Description
!=	Not equal	$x \neq y$ returns TRUE if expression x is not equal to expression y ; otherwise, it returns FALSE.

The first relational operator is the Equal operator. Note that this is exactly the same as the assignment operator. This means that the equal sign (=) is overloaded. In some circumstances, it assigns a value to a variable and other times it compares the equality of two expressions and returns TRUE or FALSE depending on the outcome of the comparison. The rule for this is that wherever InstallScript expects a relational expression, it treats the equal sign as a relational operator. This means that you cannot assign a value to variable and, at the same time, check to see if the expression of which this variable is a part is equal to some other value.

The following example examines a typical assignment statement that is used in C programming and to see what happens in InstallScript. Figure 7-3 shows the code for this simple program that attempts to use a single line of code to set three variables to the same value.

When you declare an integer variable, the variable is automatically initialized to zero. The approach used by the InstallScript engine to evaluate the assignment statements shown in the Figure 7-3 code is as follows:

```
a = (b = (c = 0));
```

First the relational expression $(c = 0)$ is evaluated and found to be TRUE so this relational expression is set to TRUE which is the same as 1. Then the relational expression $(b = 1)$ is evaluated. Since b was initialized to 0, this expression returns FALSE, which is 0. Finally the variable a is assigned the value of 0 because the expression $(b = 1)$ evaluated to FALSE. The same process can be followed to see that for the statement

```
d = e = f = 1;
```

the variable `d` is assigned the value of 1.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates the results
//                of mixing arithmetic and relational operators.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION   "Feedback"

INT a, b, c;
INT d, e, f;

program

    // Typical operations in C language.
    a = b = c = 0;
    d = e = f = 1;

    // Print the results of the assignment statements.
    SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
        "a = %d, b = %d, c = %d\n\nd = %d, e = %d, f = %d",
            a, b, c, d, e, f);

endprogram

```

Figure 7-3: *Setup.rul to demonstrate the mixture of assignment and equality operators.*

When arithmetic expressions are combined with relational expressions, it results in a value of `TRUE` or `FALSE`. Since the arithmetic operators have a higher precedence than the relational operators, the evaluation of the arithmetic expressions occurs before the evaluation of any relational operators. This is demonstrated in the next program (Figure 7-4). This program contains a statement that checks to see if the remainder of the value of a variable divided by 2 is zero or not. If the result is equal to zero, the value of the variable is an even number and if it is not equal to 0, the value of the variable is an odd number.


```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates the
//                 combination of an arithmetic expression
//                 and a relational expression.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION    "Feedback"

INT      iVal;
BOOL     bIsEven;

program

    iVal = 5;

    // Check to see if the remainder of
    // iVal when divided by 2 is 0.
    bIsEven = iVal % 2 = 0;

    // Check if iVal is an even or odd number.
    if(bIsEven) then
        sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
                  "iVal is an even number");
    else
        sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
                  "iVal is an odd number");
    endif;

endprogram

```

Figure 7-4: *Setup.rul demonstrating the combination of an arithmetic and relational expression.*

In the program shown in Figure 7-4, the output displayed in the message box is the string "iVal is an odd number". If you use any type of relational expression in a statement, the result of that statement is either TRUE or FALSE.

The next paragraphs discuss the logical operators that allow you to combine relational expressions into more complex statements and conditions. The logical operators are shown in Table 7-3.

Table 7-3: The Logical Operators

Operator	Action	Discussion
<code>&&</code>	AND	<code>x && y</code> evaluates to TRUE if both expressions <code>x</code> and <code>y</code> are TRUE; otherwise, it evaluates to FALSE.
<code> </code>	OR	<code>x y</code> evaluates to TRUE if either or both expression <code>x</code> or expression <code>y</code> are TRUE; otherwise, it evaluates to FALSE.
<code>!</code>	NOT	<code>! x</code> evaluates to TRUE if expression <code>x</code> is FALSE and evaluates to FALSE if expression <code>x</code> is TRUE.

One of things that to notice about the logical operators shown in Table 7-3 is that there is no exclusive OR operator. You can create an XOR capability by using the built-in logical operators. The code for this is provided in Figure 7-5.

This program uses the `bTrueArg` and `bFalseArg` variables to represent the evaluation of some relational expression. It then uses these variables to create all possible scenarios for the XOR truth table. Using parentheses, the program performs an inclusive OR on the variables and also performs a logical AND on them as well. It then negates the result from the logical AND and performs another logical AND on the results of the operations inside the parentheses.

This construct may look confusing when you first look at it but if you work through it by hand you can see why it produces the exclusive OR of two expressions that have a Boolean result. If you find the need to have an exclusive OR capability to use in your `if`, `while`, or `repeat` statements you can create a function that has just one line of code that returns a Boolean result after passing two expressions as arguments. The one line of code would be a return statement with the logical statement shown in the following program example.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates how to create
//                an exclusive OR evaluation using a combination
//                of the built-in logical operators.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION    "Feedback"

BOOL    bTrueArg, bFalseArg;
BOOL    bResult1, bResult2, bResult3, bResult4;

program

    bTrueArg = TRUE;
    bFalseArg = FALSE;

    // Create the XOR truth table.
    bResult1 = (bTrueArg || bFalseArg) && !(bTrueArg && bFalseArg);
    bResult2 = (bTrueArg || bTrueArg) && !(bTrueArg && bTrueArg);
    bResult3 = (bFalseArg || bTrueArg) && !(bFalseArg && bTrueArg);
    bResult4 = (bFalseArg || bFalseArg) && !(bFalseArg && bFalseArg);

    // Display the XOR truth table.
    SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
        "The XOR Truth Table\n\n      %d      %d      %d\n\n" +
        "      %d      %d      %d\n\n" +
        "      " %d      %d      %d\n\n" +
        "          %d      %d      %d",
        bTrueArg, bFalseArg, bResult1,
        bTrueArg, bTrueArg, bResult2,
        bFalseArg, bTrueArg, bResult3,
        bFalseArg, bFalseArg, bResult4);

endprogram

```

Figure 7-5: *Setup.rul demonstrating the creation of an XOR functionality.*

The result is the exclusive OR. As shown in Figure 7-6, a TRUE is returned as long as one of the variables is TRUE but not both of the variables are TRUE.

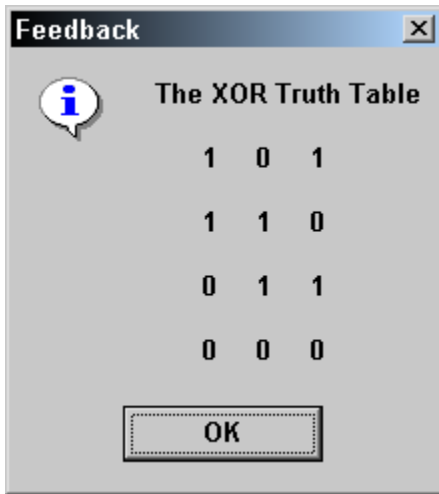


Figure 7-6: The result of the program in Figure 7-5 showing the XOR truth table display.

As with arithmetic expressions, there is precedence to the relational and logical operators. This precedence is shown below from highest to lowest.

Highest	!
	>, >=, <, <=
	=, !=
	&&
Lowest	

As with arithmetic expressions, you can control the natural order in which relational and logical expressions are evaluated (Figure 7-7). The program in Figure 7-7 adds parentheses around the expression to the left of the OR operator for the value of `bResult2`. Adding these parentheses changes the result for the total expression from `FALSE` to `TRUE`.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates how to create a compound
//                relational and logical expression.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION    "Feedback"
INT      a, b, c, d;
BOOL     bResult1, bResult2;
STRING   szResult1, szResult2;

program

    a = 2;
    b = 3;
    c = 4;
    d = 5;

    szResult1 = "FALSE";
    szResult2 = "FALSE";

    // Compound relational and logical expressions.
    bResult1 = !(a - b = c - d) && (a + b > c + d) ||
                (a * b = c * d);
    bResult2 = !((a - b = c - d) && (a + b > c + d) ||
                (a * b = c * d));

    if(bResult1) then
        szResult1 = "TRUE";
    endif;

    if(bResult2) then
        szResult2 = "TRUE";
    endif;

    // Display the results of the compound
    // relational and logical expressions.
    SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
        "bResult1 = %s\nbResult2 = %s", szResult1, szResult2);

endprogram

```

Figure 7-7: *Setup.rul showing the use of compound relational and logical expressions.*

String Expressions

In InstallScript, you can use strings to display information and to define both source paths and destination paths. A string is an array of characters and, as in the C language, you can use many functions to manipulate strings. Because of the importance of strings in defining paths, InstallScript has an additional set of functions that are used to manipulate strings that are used as paths. This section looks primarily at the operators that work with strings. String-related functions are covered in Chapter 8.

Table 7-4 describes the four special operators that help you work with strings more efficiently.

Table 7-4: The String Operators

Operator	Action	Description
+	Concatenate	This operator concatenates two strings.
^	Append to path	This adds a string to a path and, if there is no ending backslash, adds the backslash between the two strings or removes extra backslashes to maintain a valid path format.
%	Find string	This searches for one string in another string and returns TRUE if it is found or return FALSE otherwise. This operator is case insensitive.
@	String table ID	This operator in front of a string ID in the string table will bring into the script the value pointed at by the string ID.

The concatenation operator is an overloaded plus (+) sign and it allows you to add two strings together. The use of this operator has been demonstrated in some of the previous programs. The "append to path" operator is another concatenation operator

but it works with strings that are representing locations on a file system. This operator automatically includes the backslash if it is required.

The "find string" operator is used to verify that a sub-string exists in another string. The only information received from this operator is a return value of TRUE or FALSE, depending on whether the sub-string exists. To find the location of the sub-string in the string you can use the `StrFind` built-in function.

The final operator allows you to extract a string value from the string table by combining the string ID and the (@) symbol. In InstallScript, a string ID is identified by the @ symbol in front of the string ID. At run time, the InstallScript engine replaces this identifier with the string value from the string table.

InstallShield Developer makes it easy for you to add strings to your script by providing a Select String dialog (Figure 7-9). To use the dialog:

1. When your cursor is in the Script Editor, select Insert from the InstallScript drop-down menu.
2. From the sub-menu, select String Table Entry.
3. In the dialog, select the string ID you want to insert in the script.
4. Click OK. The string ID is inserted into the script at the location of your cursor and the @ symbol is placed directly in front of the string ID.

Figure 7-8 shows a program that demonstrates each of the string operators.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates the use of the
//                string operators.
//
////////////////////////////////////
#include "ifx.h"

#define CAPTION    "Feedback"

```

Figure 7-8: *Setup.rul that demonstrates the usage of the string operators.*

```

#define PATH1      "C:\\Program Files\\InstallShield\\"
#define PATH2      "C:\\Program Files\\InstallShield"
STRING  szFirst, szLast;
STRING  szPath1, szPath2;
STRING  szProductName, szResult, szFormat, szStrTable;
BOOL    bFound;

program

    szFirst = "Developer";
    szLast = "Art";

    // Use concatenation operator to create product name.
    szProductName = szFirst + " " + szLast;
    // Use the append path operator to create a path string
    // and note how the back slash is added if it is not
    // already present in the string.
    szPath1 = PATH1 ^ szProductName;
    szPath2 = PATH2 ^ szProductName;

    // Use the find string operator to verify that the
    // string Art is in szProductName.
    bFound = szProductName % szLast;

    // Use the result of the find operation to set
    // the display string. Note the use of the
    // escape character for showing a double quote
    // in a string.
    if(bFound) then
        szResult = "Substring \"%s\" was found\n\n";
    else
        szResult = "Substring \"%s\" was not found\n\n";
    endif;

    // Get value of a string ID
    szStrTable = @ID_STRING1;

    // Create the format string and then use it in the call
    // to the sprintfBox function.
    szFormat = "Product Name = %s\n\n" + "Path1 = %s\n\n" +
               "Path2 = %s\n\n" + szResult +
               "String table value = %s";

    // Display the results of the string operators.
    sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
               szFormat, szProductName, szPath1, szPath2, szLast,
               szStrTable);

endprogram

```

Figure 7-8: *Continued.*

The above program creates the string variable `szProductName` with a space between the first and last word by placing a space inside double quotes and including it by using the concatenation operator. It then creates two path-related string variables by appending the `szProductName` string variable to the `PATH1` and `PATH2` string constants defined by the `#define` statements. The append-to-path operator takes the `szProductName` string variable and appends it to both of the path string constants. The result is the same because the append-to-path operator recognizes if it needs to add the backslash or not.

By using the find string operator, the program verifies that the sub-string "Art" is in the string variable `szProductName`. This sets the `bFound` variable to `TRUE`. Using an `if` statement, the program then creates the appropriate string to display the results of this find operation. Finally, the string ID operator is used to display the value of the string ID `ID_STRING1`.

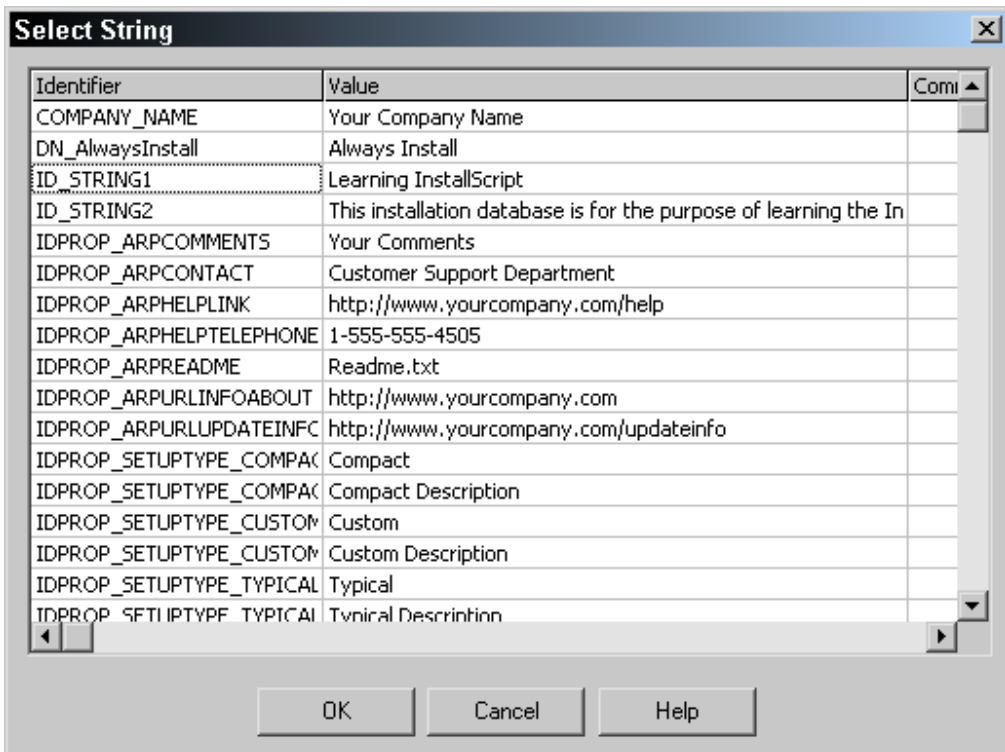


Figure 7-9: *The Select String dialog for use with the Script Editor.*

Next, the program creates a format string that will be used to display the results of all the string operations. This format string is used in the call to the `SprintfBox` function instead of entering the required format directly in the function call. In creating the format string, the program uses several escape characters that are used to display the string correctly. In strings, escape sequences allow you to display special characters that would otherwise mean something different when the string is being displayed. The escape sequences that `InstallScript` supports are shown in Table 7-5.

Table 7-5: The Escape Sequences

Code	Description
<code>\n</code>	This inserts a carriage return and a line feed.
<code>\r</code>	This inserts a carriage return and a line feed. This produces the same result as the <code>\n</code> code.
<code>\t</code>	This inserts a horizontal tab into a string.
<code>\'</code>	This inserts a single quote in a string.
<code>\"</code>	This inserts a double quote in a string.
<code>\\</code>	This inserts a backslash in a string.
<code>\ooo</code>	This inserts the ANSI character represented by the octal number <code>ooo</code> .

In the program shown in Figure 7-8, the `\n` code is used to display the output on different lines in the message box. The `\"` code is used to insert double quotes around the string value of the `sZLast` variable.

Bitwise Expressions

The ability to perform bitwise operations comes from the C language, which was originally created to replace assembly language for developing systems applications.

Accordingly, the C language was designed to provide programmers access to the computer's memory. Since C++ is the language in which InstallScript was developed, you can gain this access to the computer's memory in your InstallScript programs. There are six bitwise operators that you can use to create bitwise expressions (Table 7-6).

Table 7-6: The Bitwise Operators

Operator	Action	Description
&	AND	$x \& y$ compares the corresponding bits in x and y . The result at each position will be 1 if both bits are 1, and 0 otherwise.
	OR	$x y$ compares the corresponding bits in x and y . The result at each position will be 1 if either or both bits are 1, and 0 otherwise.
^	Exclusive OR (XOR)	$x \wedge y$ compares the corresponding bits in x and y . The result at each position will be 1 if either but not both bits are 1, and 0 otherwise.
~	One's complement (NOT)	$\sim x$ creates the one's complement of x , which means that all bits are reversed. This is the same thing as negating the value of x .
>>	Shift right	$x \gg n$ shifts the bits in x n locations to the right.
<<	Shift left	$x \ll n$ shifts the bits in x n locations to the left.

In Chapter 6, the program in Figure 6-13 used the exclusive OR bitwise operator to swap characters without having to use a temporary variable. An interesting attribute of the exclusive OR bitwise operator is that any variable that is XOR'd with itself is set to zero.

The AND bitwise operator and the OR bitwise operator have a very important functionality. The AND bitwise operator can be used to create a masking operation. This is because any bit that is 0 in either operand is set to 0 in the result. An AND bitwise operation preserves only the bits where the mask has a 1 value. The OR bitwise operator can be used to combine separate attributes into one value. This is what is done when the value to be placed in THE Attributes column of the Component table is generated. The program in Figure 7-10 shows the use of the AND and the OR bitwise operators to evaluate and modify the value in the Attributes column of the Component table. The program in Figure 7-10 uses an integer array to hold all possible attributes for the Component table. Note that the program uses hexadecimal notation to set the attributes' values. String arrays hold the attributes' descriptions. The `iAttributeValue` integer variable holds the attributes that are set in the Component table. To check to see which of the first three attributes is set the program uses an `if` statement. Because these attributes are exclusive, only one of them can be set at a time. The program loops through the remainder of the attributes to evaluate which have been set. The attribute value serves as a mask and when the program bitwise ANDs this attribute with the `iAttributeValue` variable, a non-zero result occurs if the attribute has been set and a zero value if the attribute has not been set.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates the use of
//                 the bitwise AND and OR operators.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION    "Feedback"

```

Figure 7-10: *Setup.rul* for demonstrating the use of the AND and OR bitwise operators.

```

INT      iAttributes(10), i, j, iSize, iAttributeValue;
STRING  szAttribDesc(10), szDescription, szTitle;
program
    // Set up attribute arrays with one array to hold the
    // bit-flag and the other array to hold the description.
    // The value of the attributes taken from the Windows
    // Installer help file for the Component table.
    iAttributes(0) = 0x0000;
    iAttributes(1) = 0x0001;
    iAttributes(2) = 0x0002;
    iAttributes(3) = 0x0004;
    iAttributes(4) = 0x0008;
    iAttributes(5) = 0x0010;
    iAttributes(6) = 0x0020;
    iAttributes(7) = 0x0040;
    iAttributes(8) = 0x0080;
    iAttributes(9) = 0x0100;

    // Array of strings to hold attribute description.
    szAttribDesc(0) = "Local Only";
    szAttribDesc(1) = "Source Only";
    szAttribDesc(2) = "Optional";
    szAttribDesc(3) = "Registry Key Path";
    szAttribDesc(4) = "Shared DLL Reference Count";
    szAttribDesc(5) = "Permanent";
    szAttribDesc(6) = "ODBC Data Source";
    szAttribDesc(7) = "Transitive";
    szAttribDesc(8) = "Never Overwrite";
    szAttribDesc(9) = "64-bit Component";
    // Set size of the array
    iSize = SizeOf(iAttributes);

    // Set title in message box.
    szTitle = "Present Attributes";

    // Set the present value of the attribute.
    iAttributeValue = 8;

    // Display the attribute for both the present
    // and the revised attribute setting.
    for j=0 to 1
        // Evaluate the present attribute value.
        // The first three attributes are exclusive
        // so a special check must be performed.
        if(iAttributeValue & iAttributes(1)) then
            szDescription = szAttribDesc(1);
        elseif(iAttributeValue & iAttributes(2)) then
            szDescription = szAttribDesc(2);

```

Figure 7-10: *Continued.*

```

else
    szDescription = szAttribDesc(0);
endif;

// Check the remaining attributes
for i=3 to iSize-1
    if(iAttributeValue & iAttributes(i)) then
        szDescription = szDescription + ", " +
            szAttribDesc(i);
    endif;
endfor;

// Display the results of the string operators
sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
    "%s\n\n%s", szTitle, szDescription);

// Modify the present attribute value to include
// the permanent and transitive bit-flags.
iAttributeValue = iAttributeValue |
    iAttributes(5) | iAttributes(7);

szTitle = "Revised Attributes";
endfor;

endprogram

```

Figure 7-10: *Continued.*

For each attribute that is set, the program creates a string that includes the attribute's description. This description string is then displayed in the first message box. The program then adds some additional attributes to the `iAttributeValue` variable and runs this through the check of attributes to create a new display string. The new attributes are incorporated by using the bitwise OR operator to add them to the `iAttributeValue` variable.

The next program uses the one's complement (`~`) bitwise operator to demonstrate how a computer handles negative numbers (Figure 7-11). Negative numbers are represented by the two's complement of a number. The two's complement of a number is created by first taking the one's complement and adding 1 to that result. Subtraction is handled by first taking the two's complement of the number being subtracted and then adding it as shown in the following expression.

$$a - b = a + (-b);$$

The program in Figure 7-11 uses the two's complement to negate a number.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates the use of
//                 the bitwise one's complement operator.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION    "Feedback"

INT      a, b, c, d, e, f;

program

    // Show that the two's complement creates
    // the negative of a number.
    a = 10000;
    b = ~a + 1; // Two's complement of a.
    // Use the two's complement to subtract two numbers.
    c = 1024;
    d = 512;
    e = ~d + 1; // The two's complement of d.
    f = c + e; // This is the same as c - d.

    // Display the results of the bitwise manipulation.
    sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
        "a = %d\tTwo's complement of a = %d\n\n" +
        "The value of c - d = %d", a, b, f);

endprogram

```

Figure 7-11: *Setup.rul* that uses the one's complement bitwise operator to perform subtraction.

Before we end this section on bitwise expressions, we need to discuss the bit-shift operators. When you right-shift the bits in a number by n locations, it effectively divides the number by the n th power of 2 (2^n). If the number is negative, the sign bit will be maintained, 1's are shifted into the left end of the number, and bits are shifted off the right end of the number. If the number is positive, then right-shifting bits moves 0's in on the left. If shifting bits to the right moves off any 1's on the right,

they are lost forever and cannot be regained by shifting the bits back to the left. When bits are shifted to the left, 0's are shifted into the right side of the number. The effect of left-shifting n bits is to multiply the number by the n th power of 2 (2^n).

You can use the shift bitwise operators and the concept of a mask shown in Figure 7-10 to pack characters into an integer variable. This is demonstrated in another program (Figure 7-12). This program creates an array of four characters that are packed into an integer. For each character, the program left-shifts the characters in the `iPacked` variable by eight bits. After the shift, the program uses the OR bitwise operator to add the next character to the variable. When the program finishes, the value for the `iPacked` integer variable is `7778797a` in hexadecimal format. Each pair of hexadecimal digits represents the ANSI value of the four characters that were packed into the integer.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates the use of
//                 the bitwise operators to pack characters
//                 into an integer and then unpack them.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION    "Feedback"
#define NBITS      8

INT      iMask, iPacked, iSize, i, j;
CHAR     cChars(4), cUnpackedChars(4); //Arrays of 4 characters each.
STRING   szDisplay;

program

    // Set character array
    cChars(0) = 'w';
    cChars(1) = 'x';
    cChars(2) = 'y';
    cChars(3) = 'z';

```

Figure 7-12: *Setup.rul* that uses the *shift* bitwise operators to pack characters in an integer.


```

//Get array size
iSize = SizeOf(cChars);
// Set iPacked to first character to be stored
iPacked = cChars(0);

// Pack the other characters into iPacked
iPacked = (iPacked << NBITS) | cChars(1);
iPacked = (iPacked << NBITS) | cChars(2);
iPacked = (iPacked << NBITS) | cChars(3);

// Set j index
j = 0;

// Unpack iPacked and place characters into a display string.
for i=iSize-1 downto 0
    // Set mask
    iMask = 0xFF;
    iMask = iMask << (i * NBITS);
    // Place the unpacked characters into an array.
    // Unpack from left to right.
    cUnpackedChars(j) = (iPacked & iMask) >> (i * NBITS);
    j++;
endfor;

// Display the results of the bitwise manipulation.
SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
    "iPacked = %#0.8x\n\n" +
    "The unpacked characters\n%c, %c, %c, %c",
    iPacked, cUnpackedChars(0), cUnpackedChars(1),
    cUnpackedChars(2), cUnpackedChars(3));

endprogram

```

Figure 7-12: *Continued.*

After the program finishes packing the characters, it unpacks them by using a mask that has all 1's in the first eight bits of the mask and 0's for all other bits. The program left-shifts the mask to mask each of the characters and, using the AND bitwise operator, extracts each character in order from left to right and places them in a new character array. It then displays the unpacked characters.

Expressions Using the SizeOf and Resize Operators

Some of the programs in this chapter use the `SizeOf` operator to find an array's size and then use the result to index a loop through all array elements. The `SizeOf` operator has the following prototype:

```
NUMBER SizeOf(<variable-name>);
```

You can use this operator to find the number of elements in an array, and the number of bytes in a structure or a string. The `SizeOf` operator cannot be used to obtain the number of bytes in an array element unless you first assign the array element to another variable. It is recommended that the string function `StrLength` be used to find the number of bytes in a string. However, if a string has embedded null characters, you should use the `SizeOf` operator if you want to find the total number of bytes including the null characters. The `SizeOf` operator is not a compile-time operator like it is in the C language and it cannot be used to get the size of a data type. You must first declare a variable of the data type and then get the size of the variable. For the `VARIANT` data type the variable has to reference a structure or an array, otherwise trying to get the size results in an exception.

The `Resize` operator is used to reset the size of strings and arrays. Note, however, that it is not possible to resize a string to a size smaller than it was originally declared. For example, if you declare a string consisting of 10 characters, you cannot resize it to a string that can hold only 5 characters. For full flexibility with the string and array sizing, you should always declare the variables without specifying a size. This way, you can resize up or down as needed. The `Resize` operator has the following prototype:

```
NUMBER Resize(variable-name, size);
```

Figure 7-13 shows a simple program that uses the `SizeOf` and `Resize` operators to manipulate various variable types.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates the use of
//                 the SizeOf and Resize operators.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION    "Feedback"

INT      iSize1, iSize2, iValue, i;
STRING  szDisplay, szAlphabet, szArray(10), szElement;
VARIANT vArg;

program
    iValue = 10000;
    szAlphabet = "abcdefghijklmnopqrstuvwxy";
    szDisplay = "This is a string";

    // Get size of an integer in bytes.
    iSize1 = SizeOf(iValue);

    // Set the values for the array elements.
    for i=0 to 9
        vArg = i + 1;
        szArray(i) = szDisplay + " " + vArg;
    endfor;

    // Calculate the bytes contained in the array.
    for i=0 to 9
        szElement = szArray(i);
        iSize2 = iSize2 + SizeOf(szElement);
    endfor;

    // Resize the szAlphabet string.
    Resize(szAlphabet, 13);

    // Display the results.
    SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
        "Integer size = %d bytes\nString array size = %d bytes" +
        "\nAlphabet = %s",
        iSize1, iSize2, szAlphabet);
endprogram

```

Figure 7-13: *Setup.rul* that uses the *SizeOf* and *Resize* operators.

This program sets an integer variable to some value and then uses the `SizeOf` operator to obtain the size of the integer data type. It then loops through the elements of a string array, setting each element to a different string value. Next, the program loops through this array again, obtains the size (in bytes) of each element in the array, and sums these values to get the total number of bytes used by the string array. Finally, the program uses the `Resize` operator to change the size of the `szAlphabet` string variable that has been set to the complete letters in the alphabet. The program displays the resized string, truncated according to the size used in the `Resize` operator.

Statements

There are three types of statements in `InstallScript`: statements that select which line of code to execute based on a condition, statements that loop through a number of lines of code, and statements that jump to another part of a program. Chapter 9 introduces a special statement that is used to implement exception handling.

The selection statements consist of the `if` and `switch` statements. The statements used for looping are `while`, `for`, and `repeat`. The `goto`, `return`, `exit`, and `abort` statements are used to jump from one location to another. In the true sense, expressions are also statements. An expression statement is simply a valid expression that is terminated with a semi-colon. These types of statements are discussed in the last section. The statements discussed in this section are typically categorized as flow of control statements.

Selection Statements

The two selection statements supported in `InstallScript` are the `if...endif` and the `switch...endswitch` statements. In the following discussion, square brackets (`[]`) are used to indicate optional items.

if...endif

The general format of the simple `if` statement is as follows:

```
if[ (]expression[)] then
```

```

        statement(s);
[else]
    [statement(s);]
endif;

```

In the above, *expression* must evaluate to a value that is of the NUMBER data type. You can use a variable declared as type BOOL because it is an alias for the NUMBER data type. The parentheses around *expression* are optional, as is the use of the else statement. The if statement evaluates *expression* and, if its value is non-zero, the statements that immediately follow the if statement are executed. If *expression* evaluates to zero, the statements that immediately follow the else statement are executed. If there is no else statement, program execution begins with the first statement following the endif statement.

The statements that are the targets of either the if or the else statements can themselves be if statements. This means that, for more complex selection scenarios, you can nest if statements inside each other. The format for a nested set of if statements is as follows:

```

if[()]expression[]) then
    if[()]expression[]) then
        statement(s);
    [else] // Belongs to inside if statement
        [statement(s);]
    endif;

    statement(s);
[else] // Belongs to outside if statement
    if[()]expression[]) then
        statement(s);
    [else] // Belongs to inside if statement
        [statement(s);]
    endif; // Belongs to inside if statement

    [statement(s);]
endif; // Belongs to outside if statement

```

There is no language limit to the levels of nested if statements that you can use. However, too many levels would make the code hard to read, understand, and debug. You can use the elseif construct to eliminate the need for multiple levels of if statements. The format of this construct is as follows:

```

if[()]expression1[] then
    statement(s);
elseif[()]expression2[] then
    statement(s);
[else]
    [statement(s);]
endif;

```

You can use any number of `elseif` statements, but if your program contains many `elseif` statements, you might use the `switch` statement instead. If *expression1* for the `if` statement is non-zero, the statements immediately following it are executed. If *expression1* evaluates to zero, then *expression2* is evaluated and, if it is non-zero, the statements immediately following it are executed. If both *expression1* and *expression2* evaluate to zero, the statements immediately following the `else` statement are executed.

There is a special form of the `if` statement that combines with the `goto` statement. The format of this construct is as follows:

```
if[()]expression[] goto label;
```

This statement is used to specify a jump to another location in your program identified by a label. When *expression* evaluates to non-zero, program execution moves to the first statement following the label. If *expression* evaluates to zero, program execution continues with the statement that immediately follows the `if` statement.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates the use of
//                 the if statement in a program that calculates
//                 the date following a give date.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION    "Feedback"

```

Figure 7-14: *Setup.rul* that demonstrates the use of the `if` statement.

```
// Declare a data structure.
typedef ADATE
begin
    INT     month;
    INT     day;
    INT     year;
end;

ADATE     Today, Tomorrow;
INT       iDaysPerMonth(12);
STRING    szMonthName(12);

program

    // Set the name of month array.
    szMonthName(0) = "January";
    szMonthName(1) = "February";
    szMonthName(2) = "March";
    szMonthName(3) = "April";
    szMonthName(4) = "May";
    szMonthName(5) = "June";
    szMonthName(6) = "July";
    szMonthName(7) = "August";
    szMonthName(8) = "September";
    szMonthName(9) = "October";
    szMonthName(10) = "November";
    szMonthName(11) = "December";

    // Set the days per month array.
    iDaysPerMonth(0) = 31;
    iDaysPerMonth(2) = 31;
    iDaysPerMonth(3) = 30;
    iDaysPerMonth(4) = 31;
    iDaysPerMonth(5) = 30;
    iDaysPerMonth(6) = 31;
    iDaysPerMonth(7) = 31;
    iDaysPerMonth(8) = 30;
    iDaysPerMonth(9) = 31;
    iDaysPerMonth(10) = 30;
    iDaysPerMonth(11) = 31;

    // Define today's date.
    Today.month = 2;
    Today.day = 28;
    Today.year = 2001;
```

Figure 7-14: *Continued.*

```

// Check if the present year is a leap year and set
// the days in February accordingly.
if((!(Today.year % 4) && (Today.year % 100)) ||
    !(Today.year % 400)) then
    iDaysPerMonth(1) = 29;
else
    iDaysPerMonth(1) = 28;
endif;

if(Today.day < iDaysPerMonth(Today.month-1)) then
    Tomorrow.day = Today.day + 1;
    Tomorrow.month = Today.month;
    Tomorrow.year = Today.year;
elseif(Today.month = 12) then // Check for end of year.
    Tomorrow.day = 1;
    Tomorrow.month = 1;
    Tomorrow.year = Today.year + 1;
else // This is the end of the month
    Tomorrow.day = 1;
    Tomorrow.month = Today.month + 1;
    Tomorrow.year = Today.year;
endif;

// Display today's and tomorrow's dates.
sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
    "Today's Date is:\t%s %d, %d\n\n" +
    "Tomorrow's Date is:\t%s %d, %d",
    szMonthName(Today.month-1), Today.day, Today.year,
    szMonthName(Tomorrow.month-1), Tomorrow.day,
    Tomorrow.year);
endprogram

```

Figure 7-14: *Continued.*

The program in Figure 7-14 demonstrates the use of the `if` statement. This program calculates the date that follows a specified date. This program sets values for the two arrays and then defines the date on which the calculation will be based. The first `if` statement checks the year to see if it is a leap year and sets the number of days in February accordingly. The `if...elseif` statement calculates to determine date that follows the date defined as today's date. The program then displays both today's date and the date that follows. It uses the month to index into the `szMonthName` array to display the month's name, rather than the number.

switch...endswitch

As has already been stated, the `switch` statement is a built-in multiple-branch selection statement that serves as a replacement for the `if...endif` statement. The general format for the `switch` statement is as follows:

```
switch(expression)
    case value1 [,value2, value3,...]:
        statement(s);
    case value4 [,value5, value6,...]:
        statement(s);
    case value7 [,value8, value9,...]:
        statement(s);
    [default:]
        [statement(s);]
endswitch;
```

The value of *expression* in the `switch` statement can evaluate to any of the built-in data types including `STRING` and `VARIANT`. The value of *expression* is compared against values defined by the `case` statements and, when a match is found, the associated statements are executed. If no match is found for the value defined by any of the `case` statements, the statements after the `default` statement are executed. The `default` statement is optional. Since the statements that are the target of a `case` statement can be any valid InstallScript statement, you can nest the `switch` statement inside another `switch` statement.

The C language implementation of the `switch` statement provides a fall-through capability if a `break` statement is not used between `case` statements. A fall-through capability allows for the execution of the same code for multiple `case` statements. In InstallScript, the `break` statement is built into the language's implementation so fall-through functionality is implemented by allowing a list of values to be associated with a `case` statement. The list of values uses a comma (,) as a delimiter.

Figure 7-15 shows a small program that demonstrates the use of the `switch` statement to encode the characters of a string. In particular this program shows the use of the fall through functionality by listing the values for a particular `case` statement separated by commas.

In this simple program, the first `case` statement catches all the vowels (either lowercase or uppercase) and adds 127 to the character code. It also catches the

common punctuation by another case statement and passes these characters on without any change. The consonants are caught by the default statement and modified as shown. This could have been done using an `if...elseif` statement with conditions that would check for the vowels, consonants, and punctuation. However, the `switch` statement is easier to write and understand in this example.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates the use of
//                the switch statement in a program that encodes
//                the characters of a string.
//
////////////////////////////////////
#include "ifx.h"

#define CAPTION    "Feedback"

STRING  szOriginal, szEncoded;
INT     iLen, i;

program

    szOriginal = "This is a secret message.";
    iLen = SizeOf(szOriginal);
    Resize(szEncoded, iLen); // Set to same size as szOriginal

    while(szOriginal[i] != '\0') // Stop at end of string
        switch (szOriginal[i])
            // For all vowels change to the equivalent
            // extended ANSI character.
            case 'a', 'A', 'e', 'E', 'i', 'I', 'o', 'O', 'u', 'U':
                szEncoded[i] = szOriginal[i] + 127;
            // All spaces are changed to the left curly brace.
            case ' ':
                szEncoded[i] = 123;
            // Pass all punctuation without change.
            case '.', '?', ',', '-':
                szEncoded[i] = szOriginal[i];
            // All consonants have 33 subtracted from the ANSI code.
            default:
                szEncoded[i] = szOriginal[i] - 33;
        endswitch;

```

Figure 7-15: *Setup.rul demonstrating the use of the switch statement.*

```

        i++;
    endwhile;

    // Display original and encoded strings.
    SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
        "Original: %s\n\nEncoded: %s",
        szOriginal, szEncoded);
endprogram

```

Figure 7-15: *Continued.*

Iteration Statements

This section discusses the looping statements that allow you to iterate through a set of statements a number of times. The statements in InstallScript that support looping are `for...endfor`, `while...endwhile`, and `repeat...until`. This section's discussion begins with the `for` statement.

for...endfor

The `for` statement is used primarily to loop through a set of statements a certain number of times. It is possible to create an infinite loop using a `for` statement and to break out of a `for` loop using an `if` statement and a `goto` statement. The general format of the `for` statement is as follows:

```

for index-initialization to | downto expr1 [step expr2]
    statement(s);
endfor;

```

The *index-initialization* expression is used to set the starting value of the index that will be incremented or decremented during a `for` statement. The index of a `for` loop can be any expression that evaluates to an integer value. The `to` keyword indicates that the loop index will be incremented until it is greater than the value of *expr1*. At this point, the loop terminates. The `downto` keyword means that the value of the loop index will be decremented until it is less than the value of *expr1* and then the loop terminates. If the initial value of the loop index is greater than *expr1* when incrementing, or less than *expr1* when decrementing, the loop will not execute any statements but will jump to the first statement following the `endfor` statement.

The default increment or decrement step size is 1, but can be changed with the `step` keyword. The example above uses the `step` keyword to define `exp2`, which evaluates to an integer value to increment or decrement the loop index. If `exp2` evaluates to zero, the program creates an infinite loop that will not stop until it encounters the `goto` statement. However, you cannot define a label inside a `for` loop and then jump into the loop.

Figure 7-16 shows a program that uses the `for` statement to perform a selection sort of an array.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates the use of
//                 the for statement in a program that implements
//                 the selection sort algorithm to sort an array.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION    "Feedback"

STRING  szOriginal, szEncoded;
INT     iArray(20), iTemp, iSize, min, i, j, iVal;

program

    iVal = -1;

    // Get size of array.
    iSize = SizeOf(iArray);

    // Create array to be sorted.
    for i=0 to iSize-1
        iArray(i) = i * i * iVal;
        iVal = iVal * -1;
    endfor;

```

Figure 7-16: *Setup.rul demonstrating the for statement and the selection sort of an array.*

```

// Display original array values.
for i=0 to iSize-1
    SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
        "Original Array\n\nThe Value of Element %d is %d",
            i, iArray(i));
endfor;

// Perform the selection sort of the array.
for i=0 to iSize-2
    min = i;
    // Find the minimum value in the remaining elements
    // that have not already been sorted.
    for j=i+1 to iSize-1
        if(iArray(j) < iArray(min)) then
            min = j;
        endif;
    endfor;
    // Swap first non-sorted element with the minimum
    // value found in the search.
    iTemp = iArray(min);
    iArray(min) = iArray(i);
    iArray(i) = iTemp;
endfor;

// Display sorted array values.
for i=0 to iSize-1
    SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
        "Sorted Array\n\nThe Value of Element %d is %d",
            i, iArray(i));
endfor;

endprogram

```

Figure 7-16: *Continued.*

This program first creates an array that is not initially sorted and then cycles through the array and displays the values to verify that it is not sorted. The implementation of the selection sort algorithm uses nested `for` loops. The outer loop performs the sort and the inner loop finds the minimum value in the remaining part of the array that is not yet sorted. After the sorting is complete, the program cycles through the array to display it in sorted order. In this case, the program displays the values one at a time because the `SprintfBox` function limits the number of arguments that can be passed.

while...endwhile

The `while` statement is useful because it does not require you to specify the number of iterations. You can create a condition that determines the number of iterations, which can be zero or more. You can also end the execution of a `while` statement using a `goto` statement to jump outside of the loop structure. This approach requires a condition to execute the `goto` statement at the appropriate time. The general format of the `while...endwhile` statement is as follows:

```
while [(] condition [)]
    statement(s);
endwhile;
```

The parentheses around *condition* are optional but using them makes the program more readable. It is allowed, with one exception, to include any valid statement inside the `while` loop including another `while` loop. You cannot define a label that is the target of a `goto` statement inside a `while` loop.

In Figure 7-17 is an example that shows the use of the `while` statement to calculate a specified number of prime numbers.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates the use of
//                 the while statement in a program that implements
//                 the calculation of prime numbers.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION    "Feedback"

INT      iPrimes(), iTest, iNumPrimes, i, j;
BOOL     bIsPrime, bIsRootLimit;
STRING   szDisplay;
VARIANT vArg;
```

Figure 7-17: *Setup.rul demonstrating the while statement and the calculation of prime numbers.*

```
program

// Set the number of primes to be calculated.
iNumPrimes = 20;

// Resize array to hold the desired number of prime numbers.
Resize(iPrimes, iNumPrimes);

// Set the first two prime numbers so
// the program does not have to calculate them.
iPrimes(0) = 2;
iPrimes(1) = 3;

// Initialize the array index
// and the first number to test.
i = 2;
iTest = 5;

// Use a variant to initialize the string
// that will be used to display the prime numbers.
vArg = iPrimes(0);
szDisplay = vArg;
vArg = iPrimes(1);
szDisplay = szDisplay + ", " + vArg;

// Calculate the prime numbers.
while(i < iNumPrimes)
    bIsRootLimit = TRUE;
    bIsPrime = TRUE;
    j = 1;

    // Make sure that we do not test past the
    // prime number that is the square root of
    // the number being tested.
    // Also test the number to see if it is divisible
    // by a previously discovered prime number.
    while(bIsPrime && bIsRootLimit)

        // Test to make sure that we have not past
        // the square root of the number being tested.
        if((iTest/iPrimes(j)) < iPrimes(j)) then
            bIsRootLimit = FALSE;
        endif;
```

Figure 7-17: *Continued.*

```

        // Test to make sure that the test number
        // is not evenly divisible by a prime number.
        if(!(iTest % iPrimes(j))) then
            bIsPrime = FALSE;
        endif;

        j++; // Increment the internal loop index.
    endwhile;

    // If the test number is prime then add it to the array
    // and add it to the display string then increment
    // the outer loop index.
    if(bIsPrime) then
        iPrimes(i) = iTest;
        vArg = iPrimes(i);
        szDisplay = szDisplay + ", " + vArg;
        i++;
    endif;

    // Set the test number to the next odd number.
    iTest = iTest + 2;
endwhile;

// Display prime numbers.
sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
    "The first %d prime numbers:\n\n%s", iNumPrimes, szDisplay);

endprogram

```

Figure 7-17: *Continued.*

This program creates an array for a specified number of prime numbers. To make the program more efficient, the program sets the first two prime numbers. To make testing as efficient as possible, the program checks only odd numbers. This is because, other than the number two, an even number cannot be prime. The program makes use of the fact that any non-prime number can be expressed as some multiple of prime factors. Because of this property, the program only has to see if the test numbers are evenly divisible by the prime numbers already calculated. It has to check the test number only up to a value that is less than or equal to the square root of the number that is being tested. When the test discovers that a value is a prime number, it adds the value to the array of prime numbers. It uses a variable of type VARIANT to convert the number to a string so it can be concatenated it to the variable szDisplay that is used to display the prime numbers calculated.

repeat...until

The `repeat` statement differs from the `while` statement in two respects. First, it checks the condition at the end of the loop. Therefore, the statements in the loop will be executed at least once. Second, the condition required to stop the iteration is the opposite of the condition for the `while` loop. The condition of the `while` loop needs to evaluate to `FALSE` in order for the iteration to stop. The condition for the `repeat` statement needs to evaluate to `TRUE` to stop the iteration.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates the use of
//                the repeat statement in a program that
//                implements the conversion of a decimal number
//                to another base.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION    "Feedback"

STRING  szDigits(16), szDisplay;
INT     iConverted(), i, iNumber, iBase, iSize, iDisplay;

program

    // Initialize the digit array.
    szDigits(0) = "0"; szDigits(1) = "1"; szDigits(2) = "2";
    szDigits(3) = "3"; szDigits(4) = "4"; szDigits(5) = "5";
    szDigits(6) = "6"; szDigits(7) = "7"; szDigits(8) = "8";
    szDigits(9) = "9"; szDigits(10) = "A"; szDigits(11) = "B";
    szDigits(12) = "C"; szDigits(13) = "D"; szDigits(14) = "E";
    szDigits(15) = "F";
    iNumber = 128362; // Number to be converted.
    iDisplay = iNumber; // Save number for display purposes.
    iBase = 16; // New base for number.
    i = 0;

```

Figure 7-18: *Setup.rul demonstrating the repeat statement for converting a number to another base.*

```

// Convert number to the desired base.
repeat
    Resize(iConverted, i+1);
    iConverted(i) = iNumber % iBase;
    i++;
    iNumber = iNumber/iBase;
until(iNumber = 0);

// Initialize the display variable
iSize = SizeOf(iConverted);
szDisplay = szDigits(iConverted(iSize-1));

// Create the display variable
for i=iSize-2 downto 0
    szDisplay = szDisplay + szDigits(iConverted(i));
endfor;

// Display the converted number.
SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
    "The number %d in base %d is %s",
    iDisplay, iBase, szDisplay);

endprogram

```

Figure 7-18: *Continued.*

The program in Figure 7-18 shows the use of the `repeat` statement to convert a decimal number into another specified base.

This program uses the `repeat` statement because of the possibility of specifying the number to be converted as 0. This would not have worked with `while` statement since the condition would have had to be specified as `iNumber != 0` and the loop would not have executed even once. Note that the program uses a space-saving technique of placing more than one statement of code on the same line.

The basic approach to converting a decimal number to another base is to first take the modulus of the number by the new base and store this value in an array. The program then sets a new value for the number to be converted by dividing it by the base. Since it is unknown how large an array is required to store the digits for the converted number, the program resizes the `iConverted` array each time it enters the loop. This prevents having to guess an arbitrary initial size for the array. The program continues this process until the value of the number to be converted is reduced to zero.

Jump Statements

As defined for InstallScript, jump statements move program execution from one part of the program to another or they terminate the installation program in some way. This section first discusses the statement that allows you to move execution from one point in the code to another.

goto

Though the `goto` statement does not have a good reputation in the programming world, it serves several important purposes in InstallScript. The format of this statement is as follows:

```
goto Label;
```

```
Label:
```

`Label` is a valid identifier that is terminated with a colon (:). When the `goto` statement is executed, program execution jumps to the first valid statement following `Label` and continues from that point. `Label` can be placed after or before the `goto` statement. When placing `Label` before the `goto` statement, you need to be careful not to create an infinite loop. As mentioned earlier, a `goto` statement is the only way to break out of a loop prior to the loop's normal termination. Another more important use of the `goto` statement is to enable the implementation of a user interface wizard in a Standard project. The subject of creating a user interface is covered in Chapter 12

One final thing with regard to the `goto` statement and that is you cannot place a label immediately prior to the `end` statement of a function or the `endprogram` statement in an explicit program block. The way to bypass this restriction is to place an empty statement just after the label. An empty statement consists of a single semicolon (;) without any other code.

return

The `return` statement terminates a function's execution and optionally returns a value from the function. The general format of the `return` function is as follows:

```
return [(] value [)];
```

You can return a value of any of the supported data types in InstallScript. You can also terminate a function's execution by using the `return` statement without specifying a return value if the function has been declared as having a `VOID` return type. Functions are discussed in Chapter 8.

exit and abort

Both the `exit` and `abort` statements are used to terminate an installation. The difference is that the `exit` statement is used to terminate an installation under normal circumstances. This usually means that the installation has been completed successfully. The `abort` statement is used for an abnormal termination, such as when the end user cancels the installation while it is running or there is an installation error that prevents the installation from completing successfully.

Conclusion

This chapter has covered some of the main building blocks for creating InstallScript programs. Expressions, if terminated with a semi-colon, are also statements and these are expression statements. As in most programming languages there are many different types of expressions. These expression types include arithmetic expressions, logical and relational expressions, bitwise expressions, and string expressions.

There are also three different types of statement constructs in InstallScript. These are statements that allow you to make decisions and then select a specific operation to execute, statements that allow you to loop through a set of statements some number of times, and statements that allow you to either jump to different parts of the program or to jump out of the program altogether.

Chapter

8

Functions

When you write real-world applications, all the code that you write is contained in functions. Functions are the building blocks of installation programs. The function provides the mechanism for producing programs that are easy to write, read, understand, debug, maintain, and modify.

Using functions allows you to decompose a complex problem into small manageable, understandable pieces. InstallScript uses four different categories of functions: built-in functions, event handler functions, user-defined functions, and dynamic link library functions. Each of these categories of functions is different and requires different programming approaches.

Our discussion of functions begins with some basic concepts that are applicable to all functions.

Function Basics

Any function in InstallScript can call any other function. When a function is called, it is being *invoked*. When a function invokes a function, it normally passes information to the invoked function and the invoked function returns information to the invoking function. It is possible, however, to have an invoked function that does not require any information to be passed to it and might not need to return information to the invoking function. All combinations of information in both directions are possible.

When a function passes information to another function, it passes arguments to the receiving function. The function that is invoked receives the information in its formal parameters. InstallScript needs to know the name of any function that is going to be called in a script and it also needs to know the data types of the arguments that will be passed to the function. Describing this information is called *prototyping* a function. The prototype format for each of the InstallScript-supported function categories differs depending on the function category.

When you use the built-in functions, you do not have to prototype them because the prototyping is handled in the `ifx.h` header file that you include at the top of your script. This header file is included by default when you create a project. Note that the standard approach to prototyping your own functions or those that are used from dynamic link libraries should be done in your own header files. You include these header files as part of your script.

When passing arguments to a function, you can either pass the value of the argument to the function or pass a reference to the argument value. When you pass an argument by value, the argument is copied into the formal parameter of the invoked function. Changes made by the function on the argument value do not affect the value of the argument in the invoking function. When you pass an argument by reference to a function, the address of the argument is copied into the formal parameter of the invoked function. The value of the argument is accessed by the invoked function using the address that was passed. In this case, any changes made by the invoked function on this parameter are also seen in the value of the argument in the invoking function.

By default, all arguments passed to functions are passed by value unless the prototype is modified to force passing by reference. There are two keywords in InstallScript that are used to indicate the method by which arguments are being passed to a function.

These keywords are `BYVAL` and `BYREF`. Use of the `BYVAL` keyword is not required but it does make your code more understandable.

In functions, you can declare variables of any of the supported data types. When you declare a variable in a function, this variable is a local variable. All formal parameters defined in a function's prototype are local variables. A local variable declared inside a function is not accessible from any code outside the function. The only variables that can be seen by all functions are those that are declared in the declaration block that precedes the `program...endprogram` block.

In InstallScript a function can call itself. Such a function is *recursive*. Recursion is the process of defining something in terms of itself. The use of recursion can be difficult to debug because it is hard to trace the flow of the code. Recursion is in many areas, however, used to perform sophisticated sorting operations. You can use recursion in InstallScript when you create user-defined functions.

The Built-In Functions

Built-in functions are those functions that have been defined in the InstallScript engine. Currently, there are 288 built-in functions that you can use to create your installation programs. You do not have to prototype the built-in functions because that is handled by the header file `ifix.h`. One of the header files included by `ifix.h` is the header file `isrt.h` which includes a number of header files that prototype the built-in functions.

Built-In Function Prototypes and Definitions

Chapter 4 covered the location of all the InstallScript header files, but as a reminder these header files are in various folders under the following location:

```
C:\Program Files\InstallShield\Developer\Script\isrt\Include
```


As an example of how to prototype a function, examine the prototype of the `CopyFile` function that is located in the `Files.h` header file. The prototype of this function is as follows:

```
external prototype CopyFile(BYVAL STRING, BYVAL STRING);
```

The external keyword indicates that the definition of this function is found in a script library. The script library that contains the implementation of this function is found in the following location:

```
C:\Program Files\InstallShield\Developer\Script\isrt\lib\Isrt.obl
```

The prototype keyword is always required, as is the name of the function being defined. If the function has any formal parameters, you need to declare their data types, but you do not need to provide the names of the formal parameters. In the above example, each of the arguments that are passed to the `CopyFile` function must be of the `STRING` data type. The `BYVAL` keyword indicates that the arguments are to be passed by value.

The names of the formal parameters are specified when the function is defined. The definition of the `CopyFile` function is as follows:

```
// ----- CopyFile ----- //
function CopyFile(szSrcFile, szTargetFile)
begin
    if (!PthIsAbsPath(szSrcFile)) then
        szSrcFile = SRCDIR ^ szSrcFile;
    endif;

    if (!PthIsAbsPath(szTargetFile)) then
        szTargetFile = TARGETDIR ^ szTargetFile;
    endif;

    return ISRT._FileCopy(__hContext, szSrcFile, szTargetFile,
                          COMP_NORMAL);
end;
```

The main things to note about the definition of the `CopyFile` function are the function keyword and the use of the `begin` and `end` keywords. The function keyword tells the compiler that you are defining a function. The `begin` and `end` keywords define the beginning and the end of the code that defines the function. In this example, the `CopyFile` function does not declare any local variables. If you needed local variables, then these variables would be declared between the line of code that

contains the function keyword and the begin statement. Details of prototyping script functions will be covered in the section on user-defined functions.

Because functions that are defined directly in the script are linked last, you can override a built-in function's definition with your own implementation as long as the number and type of the formal parameters is the same. If you want to use the same name as a built-in function but want to have a different number and/or types of formal parameters, then you have to go into the header file where the built-in function is prototyped and comment out the applicable line of code.

Built-in Function Categories

The built-in functions in InstallScript are sub-divided into 22 categories. A number of functions appear in more than one category. This chapter does not spend much time on built-in functions because they are fully documented in the online Language Reference that comes with InstallShield Developer. A brief description of the built-in function categories follows:

Batch File: The functions in this category are a holdover from the time when modifying AUTOEXEC.BAT during an installation was a common task. These functions come in the easy (EZ) version or the advanced version sub-categories. The EZ version of the batch file functions assumes that it is the AUTOEXEC.BAT that is being modified and these functions open the file, make the requested change, and then close the file. The advanced functions are much more powerful and can be used to specify a batch file other than AUTOEXEC.BAT, if that is necessary. However, with the advanced functions, you have to specifically open and close files as separate operations.

Built-in Dialog Box: The 15 functions in this category display simple dialogs. These dialogs include message boxes, a welcome dialog, or dialogs that perform standard operations during an installation. These particular dialogs are now created using InstallScript, but used to be coded into the InstallScript engine.

Configuration File: The functions in this category are similar to the Batch File functions. Here the default configuration file is CONFIG.SYS. There are two sub-categories: the easy (EZ) versions of the functions and the advanced functions. The advanced functions provide more robust capability.

Custom Dialog Box: The 29 functions in this category allow you to create custom dialogs using InstallScript. Chapter 12 covers user interface creation.

Extensibility: The functions in this category allow you to extend the functionality of InstallScript by calling functions defined in dynamic link libraries. These functions also allow you to launch other processes.

Feature: The 27 functions in this category deal with the movement of files during the installation. These functions are only used in Standard projects. These functions permit the creation of a "script-created feature set" as well as those features that are part of the distribution media. Script-created features are those that are created at run time, rather than at build time. It is not possible to control the order in which features are installed using these functions. The Windows Installer controls this functionality.

File and Folder: These 27 functions deal with the creation, modification, and deletion of files and folders. You can also search for files and folders and compare one file with another.

Information: The functions in this category help you to query the target system to access the values of the target system parameters including the available hard drive space and the type of processor.

Initialization: These functions pertain to creating and modifying initialization files, as well as accessing information that is contained in these files. An initialization file does not have to have an .ini extension. It simply needs to have the structure of an initialization file, which is made up of sections and keyword-value pairs.

List: One of the data types that is discussed in Chapter 6 was the LIST data type. The functions in this category are used to create and work with variables of the LIST data type.

Long File Name: The functions in this category are used to convert between short file and long file name formats. If a function cannot handle long file names, you could use one of these functions to convert the long file name to a short file name.

Miscellaneous: This is a catchall category for functions that do not fit anywhere else.

Object: This pair of functions is used to work with the COM functionality in InstallScript. These functions create a COM object or validate a COM object that has been created.

Path Buffer: These functions provide an easy way to work with strings that represent an absolute path location or a semi-colon-delimited list of absolute path locations. They handle the intricacies of manipulating a path.

Registry: One of the most important changes an installation program makes to the target system is the creation of registry entries. These 21 functions provide the capability to add, modify, and remove entries in the registry.

Sd Dialog Box: There are 43 functions in this category. Of these, 38 provide dialog boxes for use in a user interface for a Standard project. The other five functions deal with the creation of InstallScript-based dialog boxes of your own design. These dialogs are not used in a Basic MSI project.

Shared and Locked Files: This group of functions is used to handle files that are shared between applications. Some of these functions are used to handle files that might be in use when your application is installed.

Shell: The functions in this category are used to integrate an application with the operating system. This includes the creation of shortcuts and folders in which shortcuts are to be created.

String: This group of functions is used to manipulate strings. You can search strings for sub-strings, get the length of a string, and change the case of the letters in a string. There are also special functions to deal with strings that are used to define a path.

Uninstallation: This group of functions is used to implement those actions that are required for a maintenance operation. Most of these functions will be used only when creating a Standard project.

User Interface: This group of functions is used to modify the user interface that is run during an installation. This ranges from setting the color of user interface elements to customizing the text shown in error messages.

Version Checking: The functions in this category work with files to find their versions. They also overwrite files based on the version of the two files.

The function categories are based on the task to be performed. These same categories are used in the online Language Reference, as well as in the Function Wizard. The Function Wizard, which is discussed in the next section, is used to correctly insert a built-in function into a script.

The Function Wizard

InstallShield Developer has a tool that provides ready access to all the built-in functions.

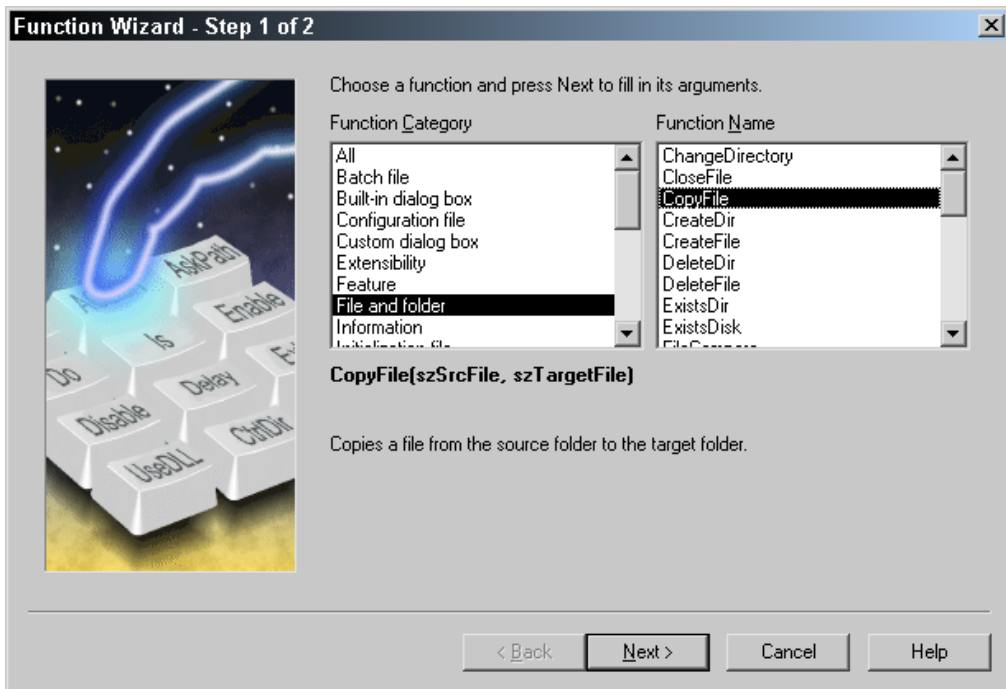


Figure 8-1: *The first panel in the Function Wizard.*

This tool is the Function Wizard and it allows you to insert a built-in function into your script with all of the proper arguments specified. The Function Wizard provides a description of the purpose of a function and a description of each of the function's arguments. In the wizard, the functions are divided into the same categories that were discussed in the last section. With your cursor in the Script Editor where you want to insert the function call, launch the Function Wizard from the InstallScript drop-down menu. From this menu, select the Insert sub-menu and then the InstallScript Function option. This option launches the Function Wizard (Figure 8-1).

The first panel displays two list boxes, one providing a list of the function categories and one providing a list of the functions in the selected category. Figure 8-1 shows that the File and folder category has been selected and the CopyFile function within that category will be inserted into the script. Below the list boxes is the prototype of the selected function and a short description of the function's purpose.

Click Next to move to the second panel (Figure 8-2).

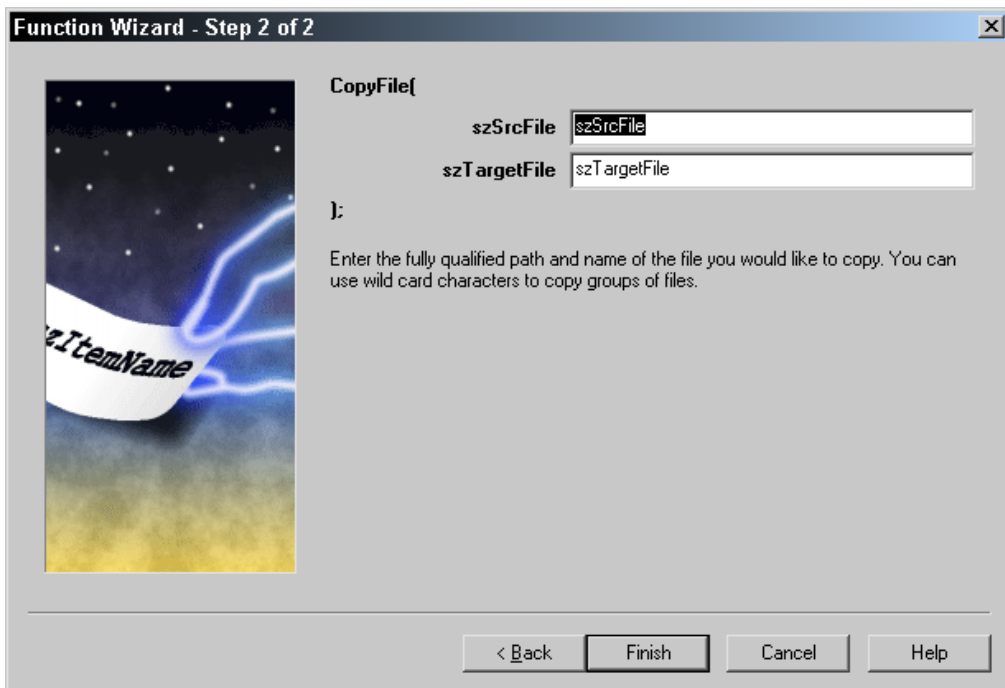


Figure 8-2: *The second panel in the Function Wizard.*

The second panel provides the capability to edit the names of the arguments that are to be passed to the selected function. By default, the fields contain the function's formal parameter names. When you put the cursor in one of the edit fields, a description of the argument appears in the panel. After you have defined the function's arguments, click Finish to insert the function call into your script (Figure 8-3). Using the Function Wizard saves time and minimizes mistakes.

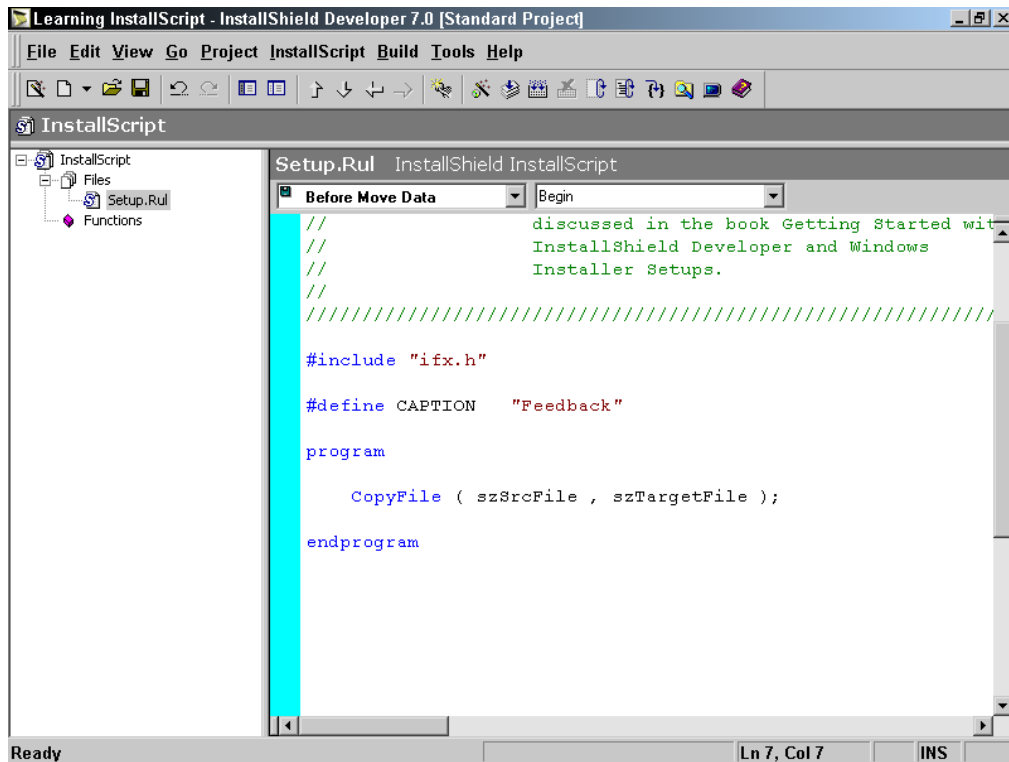


Figure 8-3: *The insertion of the CopyFile function into the script using the Function Wizard.*

Event Handler Functions

The event-driven model was described in Chapter 4. Event handler functions are functions that respond to events that are called from normal program flow logic, generated by Windows Installer engine messages, or generated by some action of the end user. It is important to understand the event-driven model so you will know

where you should place your code. All code that runs is either entered into one of the event handlers or called as a function from one of the event handlers. It is possible to bypass the event-driven model and create your own program block, but then you would have to do all of the work.

InstallScript divides the event handler functions into five major categories. These categories are described in the following list:

Before Move Data: The event handlers in this category implement actions that are preliminary to making changes to the target system. Changes to the system can mean performing a fresh install or a maintenance install. These actions include running a user interface to collect information, querying the target system for the required environment, and initializing required parameters.

Move Data: After the end user has completed running the installation's user interface, there are a number of event handlers that are called to perform operations before, after, and during the transfer of information to the target computer and/or the removal of data from the machine. By default none of these event handlers perform any operations. You need to add code to these event handlers if you need any operations carried out during a fresh install or a maintenance install.

Feature: The feature event handlers are a special type of the Move Data category. For each feature, you can specify special operations that you want to take place when that feature is being installed or uninstalled. These operations can be defined to take place before and/or after the feature is installed or uninstalled. When you define a feature event handler, the prototype of this function and an empty definition are provided in a separate .rul file called `featureevents.rul` and this file is included at the end of `Setup.rul` using the `#include` preprocessor directive.

After Move Data: The purpose of these event handlers is to provide the capability to have a user interface after the installation has finished making changes to the system. By default, an ending dialog informs the user that the installation has been completed. You could add a further user interface that would prompt the user to register the product. One of the event handlers in this category is to allow you to perform a clean up on anything that was added to the system for the purposes of performing the installation.

Miscellaneous: There are quite a few event handlers in this category that are for the purpose of handling errors and other situations that are out of the ordinary. The event handlers that are used for other types of installations other than the fresh install and the maintenance install are included in this category.

The important thing to remember about the event handler functions is that you do not have to prototype them. They are prototyped in the header file `ifx.h`. When you create a new Standard project using the Project Wizard, the `OnFirstUIBefore` and `OnMaintUIBefore` event handlers are inserted in `Setup.rul`. When you create a Standard project directly in the IDE, the `OnFirstUIBefore` and the `OnFirstUIAfter` event handlers are inserted in `Setup.rul`.

When you want to add the default script code for another event handler, first select the category of event handler. Do this at the top of the Script Editor where there are two drop-down combo boxes. The combo box on the left provides a list of the event handler categories, as well as a tree of the features defined in the project (Figure 8-4).

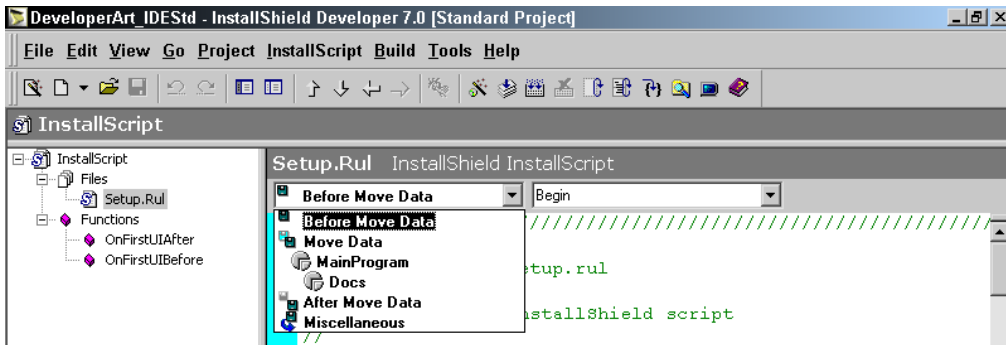


Figure 8-4: *The event handler category selection menu for the Developer Art project.*

When you select the event handler category from the menu, select the event handler that you want to add to your script from the drop-down combo box on the right (Figure 8-5).

In the combo box showing the applicable event handler functions for a certain category, the function names that appear in bold text have already been added to the script. Figure 8-5 shows that only the `OnFirstUIBefore` event handler function has been added to the script. The list of event handler functions is really a list of the type

of operations that can be implemented more than it is a true list of the event handler function names.

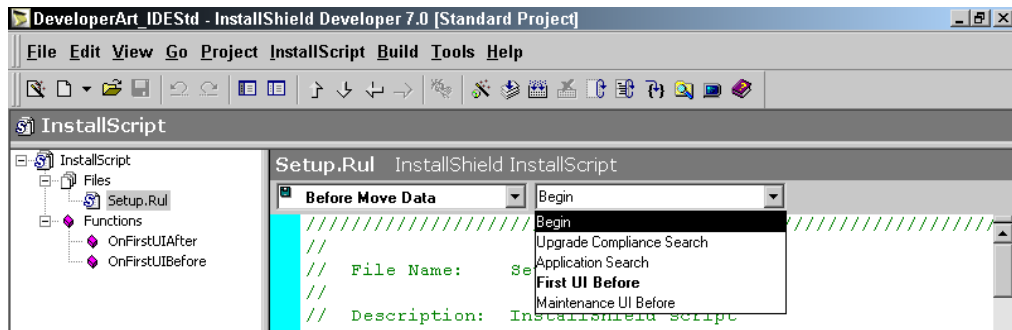


Figure 8-5: The event handler function selection menu for the Developer Art project.

When you select an event function to add to your script, the Script Editor inserts the default implementation of the function. The default implementation of all the event handler function is defined in the file `Events.rul` located in the following folder:

```
C:\Program Files\InstallShield\Developer\Support\0409
```

The Script Editor copies the default implementation from the `Events.rul` file to the script. Table 8-1 shows which event handler functions are associated with which categories. All the event handler functions are applicable to Standard setup projects. Only the `OnBegin` and `OnEnd` event handlers are supported in a Basic MSI project.

Table 8-1: Event Handler Categories and Functions

Category	Function	Description
Before Move Data	<code>OnBegin</code>	This event handler is the first one called. The default implementation is a no-op. <code>OnBegin</code> and <code>OnEnd</code> are the only two event handlers that are also supported for a Basic MSI installation.

Table 8-1: Event Handler Categories and Functions (Continued)

Category	Function	Description
Before Move Data	OnCCPSearch	This event handler verifies that, in a competitive upgrade scenario, the end user meets the criteria for the upgrade. The default implementation is a no-op.
	OnAppSearch	This event handler searches for required applications on the target system. It can also be used for searching for previous versions of the application being installed. The default implementation is a no-op.
	OnFirstUIBefore	This event handler is used to define the user interface to be displayed during a fresh install of the application.
	OnMaintUIBefore	This event handler is used to define the user interface to be displayed during a maintenance install of the application.
Move Data	OnGeneratingMSIScript	This event handler provides a chance to implement operations prior to the creation of the execution script. This event handler is executed by an immediate custom action located in the execute sequence table. The default implementation is a no-op.

Table 8-1: Event Handler Categories and Functions (Continued)

Category	Function	Description
Move Data	OnMoving	This event handler performs operations prior to making any other changes to the target system. This event handler is executed by a deferred custom action. The default implementation is a no-op.
	OnInstallFilesActionBefore	This event handler performs operations prior to copying files to the target system. This event handler is executed by a deferred custom action. The default implementation is a no-op.
	OnInstallFilesActionAfter	This event handler performs operations right after the copying of files to the target system. This event handler is executed by a deferred custom action. The default implementation is a no-op.
	OnMoved	This event handler performs operations after making changes to the target system. This event handler is executed by a deferred custom action. The default implementation is a no-op.

Table 8-1: Event Handler Categories and Functions (Continued)

Category	Function	Description
Move Data	OnGeneratedMSIScript	This event handler provides a chance to implement operations prior to the running of the execution script. This event handler is executed by an immediate custom action located in the execute sequence table. The default implementation is a no-op.
Features	<i>FeatureName</i> _Installing	This event handler performs operations prior to installing a feature with the name <i>FeatureName</i> . This event handler is executed by a deferred custom action. The default implementation is a no-op. The actual functionality is to run all these functions at the same time prior to the installation of features. The Windows Installer controls the order of feature installation.
	<i>FeatureName</i> _Installed	This event handler performs operations after installing a feature with the name <i>FeatureName</i> . This event handler is executed by a deferred custom action. The default implementation is a no-op. the operation here is the same as described above.

Table 8-1: Event Handler Categories and Functions (Continued)

Category	Function	Description
Features	<i>FeatureName_UnInstalling</i>	This event handler performs operations prior to uninstalling a feature with the name <i>FeatureName</i> . This event handler is executed by a deferred custom action. The default implementation is a no-op.
	<i>FeatureName_UnInstalled</i>	This event handler performs operations after uninstalling a feature with the name <i>FeatureName</i> . This event handler is executed by a deferred custom action. The default implementation is a no-op.
After Move Data	OnFirstUIAfter	This event handler displays a user interface after the completion of a fresh install. It needs to be recognized that once you have entered this function it is not possible to rollback an installation because the service process has finished its actions and the rollback script has been deleted.
	OnMaintUIAfter	This event handler displays a user interface after the completion of a maintenance operation. Rollback here is also impossible.

Table 8-1: Event Handler Categories and Functions (Continued)

Category	Function	Description
After Move Data	OnEnd	This is the last event handler that is called if the installation was not aborted. The OnEnd and the OnBegin event handlers are the only two events that are supported for both Standard and Basic MSI installations. The default implementation is a no-op.
Miscellaneous	OnAbort	This event handler performs operations in the case where the installation has been aborted. The default implementation is a no-op.
	OnCanceling	This event handler performs operations when the end user has canceled the installation. This event handler asks for confirmation and then calls the abort statement.
	OnComponentError	This event is called when an error occurs in launching the Windows Installer to run the actions in the execute sequence table.
	OnException	This event handler is called when an exception is raised inside a procedure-based script that includes an explicit program/ endprogram block.

Table 8-1: Event Handler Categories and Functions (Continued)

Category	Function	Description
Miscellaneous	OnHelp	This event handler is used to respond to the user pressing the F1 key or the calling of the Do (HELP) function. The default implementation is a no-op.
	OnUninstall	This event handler is triggered when the Enable Maintenance property in General Information has been set to No. This happens when a user tries to run setup.exe again after a fresh install has already been performed or the Add/Remove programs applet is used to uninstall the application. This is similar to the approach used for uninstalling legacy applications when maintenance was not an option.
	OnPatchUIBefore	This event handler displays the user interface at the beginning of a patch install for a locally installed application.
	OnPatchUIAfter	This event handler displays the user interface at the completion of a patch install for a locally installed application.

Table 8-1: Event Handler Categories and Functions (Continued)

Category	Function	Description
Miscellaneous	OnAdminPatchUIBefore	This event handler displays the user interface at the beginning of a patch install for an administrative install of an application. This user interface by default consists of just a Welcome dialog.
	OnAdminPatchUIAfter	This event handler displays the user interface at the completion of a patch install for an administrative install of an application.
	OnAdminInstallUIBefore	This event handler displays the user interface at the beginning of an administrative install.
	OnAdminInstallUIAfter	This event handler displays the user interface at the completion of an administrative install.
	OnAdvertisementBefore	This event handler is used to perform required operations prior to the running of an advertised install. The default implementation is a no-op.
	OnAdvertisementAfter	This event handler is used to perform required operations after the completion of an advertised install. The default implementation is a no-op.

Table 8-1: Event Handler Categories and Functions (Continued)

Category	Function	Description
Miscellaneous	OnMsiSilentInstall	This event handler is triggered when the end user tries to launch a silent install directly using the Windows Installer engine and not setup.exe.
	OnError	This event handler is triggered when the Windows Installer sends an error message to the external user interface.
	OnWarning	This event handler is triggered when the Windows Installer sends a warning message to the external user interface.
	OnFilesInUse	This event handler is triggered when the Windows Installer sends a message to the external user interface about the install trying to overwrite a file in use.
	OnOutOfDiskSpace	This event handler is triggered when the Windows Installer sends a message to the external user interface about there not being enough disk space.

The information shown in Table 8-1 is only a brief summary of some of what was discussed in Chapter 4. Chapter 4 provides details about how InstallShield Developer implements both Standard and Basic MSI projects.

User-Defined Functions

There are two types of user-defined functions that you can create. This applies to both Standard and Basic MSI projects. One type of user-defined function allows the function to be called by the Windows Installer engine. These types of functions are used as the targets of custom actions and they have a very strict prototype format that must be followed. This strict prototyping format makes these particular functions available as entry points in to the script. As discussed in Chapter 3, a custom action is the mechanism that can be used to extend the functionality of the Windows Installer. For a Standard project, you can use script-based custom actions only in the Execute sequence table. The other type of user-defined function is used to provide functionality that is called from event handler functions or called by the custom actions that you create. These types of user-defined functions do not have the restrictions discussed for the entry point functions. As in previous chapters, the use of square brackets when discussing prototypes denotes that the enclosed keyword is optional.

Entry Point User-Defined Functions

The prototype of an entry point functions is as follows:

```
export prototype function-name(NUMBER);
```

This prototype uses the export keyword, which identifies this function as one that will be called from outside the running script. The prototype keyword is also required but the name of the function is up to you. This type of function can be passed only one argument, which has to be of the NUMBER data type. The argument that gets passed is the handle to the running installation session. No return type is shown because this type of function needs to return a value of type NUMBER, which is the default return type for an InstallScript function. The definition of this type of function is as follows:

```
function function-name(hMSI)  
// Place declaration of local variables here.  
begin  
// Place implementation code here.  
end;
```

This example uses the `function` keyword to identify this block of code as a function. The function name used here has to be the same as used in the prototype statement. Here, in place of the data type of the formal parameter, the above example specifies the name of the formal parameter. In this case `hMSI` is used to indicate that the function is passing a handle to the install session.

You will create script custom actions in Chapter 11 where you will set up an environment for working with custom actions. Our present environment for learning InstallScript is not a good one for working with custom actions.

Generic User-Defined Functions

The format for prototyping a generic user-defined function is as follows:

```
[external] prototype [return-type]
                function-name([parameter-data-type-list], [...])
```

The `external` keyword is used to identify a function that is defined in a script library. Once again the `prototype` keyword is required and the return type for the function follows this keyword. The return type can be any of the InstallScript data types plus the designation of no return type if you use the `VOID` keyword. If you do not specify a return type, the default return type is of the `NUMBER` data type. The name of the function can be anything as long as it follows the rules for creating variable names. A generic user-defined function can have any number of formal parameters and it can specify that the number of arguments to be passed to it is undetermined at compile time. Using an ellipsis specifies that the number of arguments to be passed will be determined at run time. The designation that some of the arguments to be passed to a function will be determined at run time needs to be the last specification of formal parameters.

The definition of a generic user-defined function takes the following format:

```
function [return-type] function-name([parameter-name-list])
// Place declaration of local variables here.
begin
// Place implementation code here.
end;
```

When you prototype a function as having an unknown number of arguments the name of the formal parameter in the function declaration has to be the name of an array. You access the values being passed inside the function by getting the values contained in the array. When you pass arguments to a function, make sure that the data type of the argument passed agrees with the data type specified in the function's prototype.

Basic Example

The best approach to understanding the creation and use of generic user-defined functions is to look at some examples. The first example in Figure 8-6 shows the use of the ellipsis to send an unknown number of arguments to a function and have this function add them and return the result, as part of a string, back to the calling function.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:   Learning the InstallScript language
//
// Comments:     This script demonstrates the use of
//               a generic user-defined function for adding
//               an array of numbers and returning a string
//               describing the result of the calculation.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION    "Feedback"

// Prototype a function that will take an unknown
// number of arguments and return a string.
prototype STRING SumNumbers(...);

program

    // Display the result from the call to the SumNumbers function.
    SprintfBox ( MB_OK | MB_ICONINFORMATION , CAPTION , "%s",
                SumNumbers (1,2,3,4,5,6,7,8,9,10) );

endprogram

```

Figure 8-6: User-defined function that sums an unknown number of values and returns a string.

```

////////////////////////////////////
// Function:      SumNumbers
//
// Purpose:       This function sums the numbers that are sent to it
//                and returns a string describing the result of the
//                calculation.
////////////////////////////////////
function STRING SumNumbers(iNums)
INT      i, iSize, iSum;
STRING  szReturn;
begin

    // Get the number of arguments passed.
    iSize = SizeOf(iNums);

    // Initialize iSum to 0.
    iSum = 0;

    // Loop through the array to add up the numbers.
    for i=0 to iSize-1
        iSum = iSum + iNums(i);
    endfor;

    // Create the string to be returned by the function.
    Sprintf(szReturn, "The sum of the numbers is %d", iSum);

    return (szReturn);
end;

```

Figure 8-6: *Continued.*

The program in Figure 8-6 demonstrates the use of the ellipsis and it also shows that a user-defined function can return a string value. Since the number of values passed to the `SumNumbers` function is known only at run time, the program has to determine the size of the array holding these values before it can loop through the values and add them. In the `SumNumbers` function, note the use of the `Sprintf` built-in function to create the return value for the function. Also note that the `SumNumbers` function is called directly in the call to the `SprintfBox` function. The program does not have to create a string variable to receive the return from the function and then pass the variable to the `SprintfBox` function.

Using Recursion

Recursion is sometimes useful in creating a user-defined function. The typical example used by most programming texts to demonstrate recursion is the calculation of the factorial of an integer. This is demonstrated in Figure 8-7.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:   Learning the InstallScript language
//
// Comments:     This script demonstrates the use of
//               a generic user-defined function for using
//               recursion to calculate the factorial of a number.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION    "Feedback"

// Function to calculate the factorial of an integer.
prototype factorial(INT);

INT    iValue, iNum;

program

    // Set the number for which the factorial is calculated.
    iNum = 10;

    // Call the function to calculate the factorial.
    iValue = factorial(iNum);

    // Display the result of the factorial.
    SprintfBox (MB_OK | MB_ICONINFORMATION, CAPTION,
                "The factorial of %d is %d", iNum, iValue);

endprogram

```

Figure 8-7: *Using recursion to calculate the factorial of a number.*

```

////////////////////////////////////
// Function:    factorial
//
// Purpose:     This function calculates the factorial
//              of a number using recursion.
////////////////////////////////////
function factorial(iNumber)
INT    iResult;
begin

    // Check for the trigger to stop the recursive calls.
    if(iNumber = 1) then
        return 1;
    endif;

    // Recursively call factorial until iNumber-1 = 1
    iResult = iNumber * factorial(iNumber-1);

    return (iResult);

end;

```

Figure 8-7: *Continued.*

Using recursion is not recommended unless it is the only way to implement a certain algorithm. The important thing to remember in implementing a recursive function is to have a statement where the recursive call is prevented. If you do not have such a statement, recursion will run until there is a stack overflow. Recursion places a new copy of the local variables on the stack for each call to the function. When recursion stops, the program starts to pop these values off of the stack until it returns back to the first recursive call and then the function returns the value to the original caller. In the program in Figure 8-7, the statement that stops the recursion is the `if` statement that checks for when `iNumber = 1`.

Passing Arguments by Reference

The first example showed the return of a string value from a function through the `return` statement. Since a function can only return one value using the `return` statement, this presents a problem if you want to return more than one value. The solution is to pass one or more arguments by reference. This gives you the ability to return more than one value from a function. These values do not have to be the same data type. The example shown in Figure 8-8 shows how to return a value through an

argument that is passed by reference. This is a function-based implementation of the program given in Figure 6-13 that reverses the characters in a string.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This demonstrates the use of
//                 a generic user-defined function and the passing
//                 of an argument by reference.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION    "Feedback"

// Function to reverse the characters in a string.
prototype ReverseChars(BYVAL STRING, BYREF STRING);

STRING  szOrigStr, svRevStr;

program

    // Set the string to be reversed.
    szOrigStr = "These are the times that try men's souls.";

    // Call the function to reverse the string characters.
    ReverseChars(szOrigStr, svRevStr);

    // Print the result of the ReverseChars function.
    SprintfBox ( MB_OK | MB_ICONINFORMATION , CAPTION ,
                "Original string: %s\n\nReversed string: %s",
                szOrigStr, svRevStr);

endprogram

////////////////////////////////////
// Function:      ReverseChars
//
// Purpose:      This function reverses the characters in a string.
////////////////////////////////////
function ReverseChars(szOriginal, svReversed)
INT    i, j, iLen;
begin

```

Figure 8-8: *Returning values from a function through an argument passed by reference.*

```

// Get the length of the string to be reversed.
iLen = StrLength(szOriginal);

i = 0; // Set i to point to first string
j = iLen - 1; // Set j to point at last character

// Work from both ends of the string to
// swap the characters without using a
// temporary variable to perform the swap.
while(i < j)
    szOriginal[i] = szOriginal[i] ^ szOriginal[j];
    szOriginal[j] = szOriginal[j] ^ szOriginal[i];
    szOriginal[i] = szOriginal[i] ^ szOriginal[j];

    i++; // Increment i index
    j--; // Decrement j index
endwhile;

svReversed = szOriginal;

end;

```

Figure 8-8: *Continued.*

The key to returning a value through an argument is in the prototype of the function and the use of the BYREF keyword. This indicates that you are passing a pointer to the argument so that when the program works on what this pointer points at, it changes the value of the argument in the calling function. Note the name of the variable that returns the result of the function's operation uses the notation `sv` instead of `sz`. This type of notation appears in the online Language Reference whenever the argument to a built-in function returns a value. This book uses this notation for consistency.

The `ReverseChars` function performs the operation on the `szOriginal` formal parameter and, when the operation is finished, sets the `svReversed` formal parameter equal to `szOriginal`. The program block displays the value of both the arguments that were passed to the `ReverseChars` function. As shown in this display, reversing the characters of the `szOrigStr` argument in the function does not affect the value of this variable in the program block.

Creating a Script Library

In this section, you will learn to create a script library containing a few functions. You will then use these functions in an example program.

WRITING AND TESTING THE CODE

First, you need to create a new script file called `Sort.rul`.

1. Navigate to the InstallScript view.
2. Right-click on the Files node and select the New Script File. This creates a new script file with a default name.
3. Rename the file to `Sort.rul`.

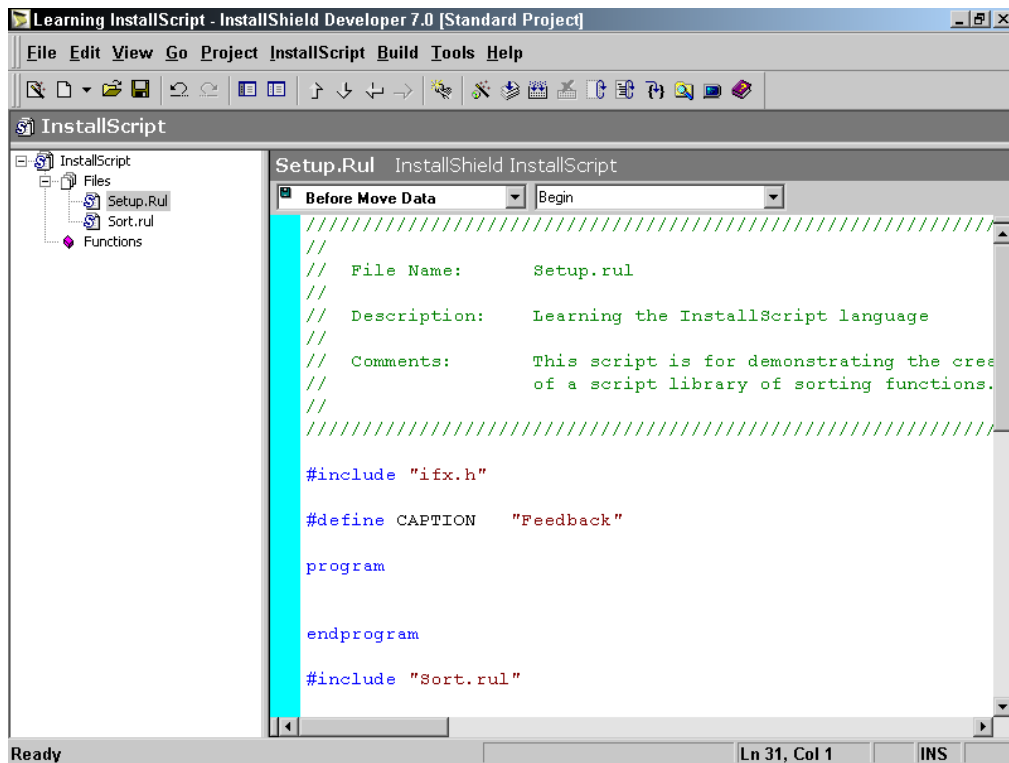


Figure 8-9: *The initial test setup for creating a script library of sorting routines.*

At the bottom of Setup.rul, use the `#include` preprocessor directive to include Sort.rul. This is to test your functions before you create the script library. In the InstallScript view, your initial setup for this activity should look like what is shown in Figure 8-9.

Note in this figure that Sort.rul is included at the bottom of Setup.rul. Also note that, at this time, there are no functions listed in the functions list to the left of the Script Editor. You will create all of your sorting functions in Sort.rul and then call them in the program block to verify that they work correctly before creating the script library. You will create four sorting routines, one that uses the selection sort algorithm, one that uses the shellsort algorithm, one that uses the quicksort algorithm, and one that uses quicksort for sorting a list.

Next, look at is the program block that you will first use to test the sorting functions (Figure 8-10). This will also be used to test the functions from the script library. Most of the code shown in Figure 8-10 is devoted to the creation of arrays and lists to use for testing, and several functions for displaying the contents of an array or a list. There are, however, a few interesting points that need to be made about this code. The first is the prototype statements used for the sorting functions that will be compiled into a script library. When you use the sorting functions from the script library, you will add the `external` keyword to the prototype statement and place them into a header file, which you will name Sort.h.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:       This script tests the functions to be
//                  built into a script library of sorting functions.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION    "Feedback"
// Prototypes of private functions.
prototype STRING CreateArrayDisplay(BYVAL VARIANT);
prototype STRING CreateListDisplay(BYVAL LIST);

```

Figure 8-10: *The program block used to test the sorting functions.*

PART II INSTALLSCRIPT

```
// Keep the external keyword commented out until using the
// the functions from the script library. When the library
// is used then these prototypes will be placed in a
// header file called sort.h and then included here.
/*external*/ prototype VOID SelectionSort(BYREF VARIANT);
/*external*/ prototype VOID ShellSort(BYREF VARIANT);
/*external*/ prototype VOID QuickSort(BYREF VARIANT, BYVAL INT,
                                      BYVAL INT);

/*external*/ prototype VOID ListQSort(BYVAL LIST);

INT      i, iVal, iSizeInt, iSizeStr, iArray(20), iValue;
STRING   strArray(10), strValue;
LIST     iList, strList;

program

    iSizeInt = SizeOf(iArray);
    iSizeStr = SizeOf(strArray);

    // Create lists for testing the list sorting function.
    iList = ListCreate(NUMBERLIST);
    strList = ListCreate(STRINGLIST);

    // Create an integer array to be sorted.
    iVal = -1;
    for i=0 to iSizeInt-1
        iArray(i) = i * i * iVal;
        iVal = iVal * -1;
    endfor;

    // Create a number list that has the same
    // values as the integer array.
    for i=0 to iSizeInt-1
        iValue = iArray(i);
        ListAddItem(iList, iValue, AFTER);
    endfor;

    // Display original integer array values.
    SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
               "Original Integer Array\n\n%s",
               CreateArrayDisplay(iArray));

    //SelectionSort(iArray);
    //ShellSort(iArray);
    //QuickSort(iArray, 0, iSizeInt-1);
```

Figure 8-10: *Continued.*

```

// Display sorted integer array values.
sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
           "Sorted Integer Array\n\n%s",
           CreateArrayDisplay(iArray));

// Create a string array to be sorted.
strArray(0) = "Feature"; strArray(1) = "Component";
strArray(2) = "Icon"; strArray(3) = "Shortcut";
strArray(4) = "Registry"; strArray(5) = "Class";
strArray(6) = "TypeLib"; strArray(7) = "File";
strArray(8) = "AppId"; strArray(9) = "TextStyle";

// Create a string list that has the same
// values as the string array.
for i=0 to iSizeStr-1
    strValue = strArray(i);
    ListAddString(strList, strValue, AFTER);
endfor;

// Display original array values.
sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
           "Original String Array\n\n%s",
           CreateArrayDisplay(strArray));

//SelectionSort(strArray);
//ShellSort(strArray);
//QuickSort(strArray, 0, iSizeStr-1);

// Display sorted string array values.
sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
           "Sorted String Array\n\n%s",
           CreateArrayDisplay(strArray));

// Display original number list values.
sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
           "Original Number List\n\n%s",
           CreateListDisplay(iList));

ListQSort(iList);
// Display sorted number list values.
sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
           "Sorted Number List\n\n%s",
           CreateListDisplay(iList));

// Display original string list values.
sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
           "Original String List\n\n%s",
           CreateListDisplay(strList));

```

Figure 8-10: *Continued.*

```

ListQSort(strList);

// Display sorted string list values.
sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
           "Sorted String List\n\n%s",
           CreateListDisplay(strList));

endprogram

/////////////////////////////////////////////////////////////////
// Function:      CreateArrayDisplay
//
// Purpose:      This function creates a display string
//               for the values in an array.
/////////////////////////////////////////////////////////////////
function STRING CreateArrayDisplay(Array)
INT      i, iSize;
STRING  szDisplay;
begin

    iSize = SizeOf(Array);

    for i=0 to iSize-1
        szDisplay = szDisplay + Array(i) + ", ";
    endfor;

    return (szDisplay);
end;

/////////////////////////////////////////////////////////////////
// Function:      CreateListDisplay
//
// Purpose:      This function creates a display string
//               for the values in a list.
/////////////////////////////////////////////////////////////////
function STRING CreateListDisplay(List)
INT      i, iSize, iType, iValue;
STRING  szDisplay, strValue;
VARIANT Value;
begin

    iSize = ListCount(List);
    iType = ListGetType(List);

    ListSetIndex(List, LISTFIRST);

```

Figure 8-10: *Continued.*

```

    if(iType = NUMBERLIST) then
        for i=0 to iSize-1
            ListCurrentItem(List, iValue);
            Value = iValue;
            szDisplay = szDisplay + Value + ", ";
            ListSetIndex(List, LISTNEXT);
        endfor;
    else
        for i=0 to iSize-1
            ListCurrentString(List, strValue);
            Value = strValue;
            szDisplay = szDisplay + Value + ", ";
            ListSetIndex(List, LISTNEXT);
        endfor;
    endif;

    return (szDisplay);
end;

#include "Sort.rul"

```

Figure 8-10: *Continued.*

The second item of interest is the creation of several lists, one a number list and one a string list. The code uses some of the built-in functions that you need to use when manipulating a variable of the LIST data type. When you pass an array to one of the sorting functions, you need to pass it using the BYREF keyword. When you pass the list to the sorting function, you need to pass it using only the BYVAL keyword. This is because the list data type is a pointer to a linked list and the sorting function makes changes to list elements and not to the pointer itself. The most important item with regard to lists is that an array is much easier to manipulate than a list and an array should be used instead of a list whenever possible.

The code for the sorting functions is shown in Figure 8-11. This is the code that you will compile, after a few changes, into a script library named Sort.obl. There are some important points to discuss regarding the code shown in Figure 8-11. The first thing to note is that the file `ifx.h` is included, but is commented out while you are in the testing mode. You will need to uncomment this `#include` statement when you build the script library. There are also prototype statements that are commented out while in testing mode. These prototype statements are the ones that are directly before the definition of the sorting functions. These prototype statements use the `export` keyword, which is necessary to make these functions available to other programs when they have been compiled into a script library.


```

////////////////////////////////////
//
// File Name:      Sort.rul
//
// Description:    Code for the sorting script library
//
// Comments:       This script demonstrates the creation
//                  of a script library of sorting functions.
//
////////////////////////////////////

// Keep this #include preprocessor statement commented out
// while testing the functions, then uncomment it when
// building the library file.
//#include "ifx.h"

// Prototypes of private functions.
prototype VARIANT GetMinArrayValue(BYVAL VARIANT);
prototype VOID Swap(BYREF VARIANT, BYVAL INT, BYVAL INT);
prototype VOID SetSentinel(BYVAL VARIANT, BYREF VARIANT);
prototype VOID RemoveSentinel(BYVAL VARIANT, BYREF VARIANT);
prototype INT ListToArray(BYVAL LIST, BYREF VARIANT, BYVAL INT);
prototype INT ArrayToList(BYVAL VARIANT, BYVAL LIST, BYVAL INT);

// Keep this prototype commented out while testing
// and uncomment it when building the library file.
//export prototype VOID SelectionSort(BYREF VARIANT);

////////////////////////////////////
// Function:      SelectionSort
//
// Purpose:       This function sorts an array
//                  using the selection sort algorithm.
////////////////////////////////////
function VOID SelectionSort(Array)
  INT      i, j, min, iSize;
  VARIANT Temp;
begin

  iSize = SizeOf(Array);

  // Perform the selection sort of the array.
  for i=0 to iSize-2
    min = i;
    // Find the minimum value in the remaining elements
    // that have not already been sorted.

```

Figure 8-11: *The sorting functions that will be compiled into a script library.*

```

        for j=i+1 to iSize-1
            if(Array(j) < Array(min)) then
                min = j;
            endif;
        endfor;
        // Swap first non sorted element with the minimum
        // value found in the search.
        Swap(Array, min, i);
    endfor;

end;

// Keep this prototype commented out while testing
// and uncomment it when building the library file.
//export prototype VOID ShellSort(BYREF VARIANT);

/////////////////////////////////////////////////////////////////
// Function:      ShellSort
//
// Purpose:       This function sorts an array
//                using the shellsort algorithm.
/////////////////////////////////////////////////////////////////
function VOID ShellSort(Array)
    INT      i, j, k, iIndex, min, iSize;
    VARIANT Temp, TempArray();
    BOOL     bCompare;
begin

    // Get size of array to sort.
    iSize = SizeOf(Array);

    // Create a temporary array with
    // a sentinel that has a value equal to
    // the minimum value in the array to be sorted.
    SetSentinel(Array, TempArray);

    // Calculate the starting value
    // for the k index.
    iIndex = 1;
    while(TRUE)
        iIndex = 3*iIndex+1;
        if(iIndex <= iSize)then
            k = iIndex;
        else
            goto EndLoop;
        endif;
    endwhile;

```

Figure 8-11: *Continued.*

```

EndLoop:

// Loop through the values of k
// starting with the highest to the lowest.
while(k > 0)
    i = k + 1;
    while(i <= iSize)
        Temp = TempArray(i);
        j = i;
        // Set the condition for the
        // inner while loop.
        if(j > k) then
            bCompare = FALSE;
            if(TempArray(j-k) > Temp) then
                bCompare = TRUE;
            endif;
        endif;
        while(j > k && bCompare)
            TempArray(j) = TempArray(j-k);
            j = j - k;
            // Set the condition for the
            // inner while loop.
            if(j > k) then
                bCompare = FALSE;
                if(TempArray(j-k) > Temp) then
                    bCompare = TRUE;
                endif;
            endif;
        endwhile;
        TempArray(j) = Temp;
        i = i + k;
    endwhile;
    k = k/3; // Redefine the value of k.
endwhile;

// Convert the sorted temporary array
// back to the array to be returned
// to the calling routine.
RemoveSentinel(TempArray, Array);

end;

// Keep this prototype commented out while testing
// and uncomment it when building the library file.
//export prototype VOID QuickSort(BYREF VARIANT, INT, INT);

```

Figure 8-11: *Continued.*

```

////////////////////////////////////
// Function:   QuickSort
//
// Purpose:    This function sorts an array
//             using the quicksort algorithm.
////////////////////////////////////
function VOID QuickSort(Array, iLower, iUpper)
  INT      i, j;
  VARIANT Partition;
  begin

    if(iLower >= iUpper) then
      return;
    endif;

    // Divide the array into two parts
    Partition = Array((iLower + iUpper)/2);

    i = iLower;
    j = iUpper;

    // Place lower values to the left and
    // higher values to the right.
    while(i <= j)
      while(i < iUpper && Array(i) < Partition)
        i++;
      endwhile;

      while(j > iLower && Array(j) > Partition)
        j--;
      endwhile;

      if(i <= j) then
        Swap(Array, i, j);
        i++;
        j--;
      endif;
    endwhile;

    // Sort each half of the array recursively.
    if(iLower < j) then
      QuickSort(Array, iLower, j);
    endif;

    if(i < iUpper) then
      QuickSort(Array, i, iUpper);
    endif;
  end;

```

Figure 8-11: *Continued.*

```

// Keep this prototype commented out while testing
// and uncomment it when building the library file.
//export prototype VOID ListQSort(BYVAL LIST);

/////////////////////////////////////////////////////////////////
// Function:      ListQSort
//
// Purpose:       This function sorts a list
//                using the quicksort algorithm.
/////////////////////////////////////////////////////////////////
function VOID ListQSort(List)
INT      iType, iReturn, iSize;
VARIANT Array();
begin

    iSize = ListCount(List);
    iType = ListGetType(List);

    iReturn = ListToArray(List, Array, iType);

    if(iReturn < 0) then
        return;
    endif;

    QuickSort(Array, 0, iSize-1);

    iReturn = ArrayToList(Array, List, iType);

end;

/////////////////////////////////////////////////////////////////
// Function:      GetMinArrayValue
//
// Purpose:       This function returns the minimum
//                value in an array.
/////////////////////////////////////////////////////////////////
function VARIANT GetMinArrayValue(Array)
INT      i, iSize;
VARIANT min;
begin

    // Get size of array
    iSize = SizeOf(Array);

    // Initialize min value of array.
    min = Array(0);

```

Figure 8-11: *Continued.*

```

    // Find if any values in the array are
    // smaller than the initial value.
    for i=1 to iSize-1
        if(Array(i) < min) then
            min = Array(i);
        endif;
    endfor;

    return (min);
end;

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Function:      Swap
//
// Purpose:       This function swaps two values in an array.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
function VOID Swap(Array, i, j)
VARIANT Temp;
begin

    Temp = Array(i);
    Array(i) = Array(j);
    Array(j) = Temp;
end;

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Function:      SetSentinel
//
// Purpose:       This function adds a sentinel value to an array.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
function VOID SetSentinel(Array, TempArray)
INT    iSize, i;
begin

    // Get size of array to sort.
    iSize = SizeOf(Array);

    // Create a temporary array that will
    // contain a sentinel value.
    Resize(TempArray, iSize+1);

    // Initialize the temporary array.
    for i=1 to iSize
        TempArray(i) = Array(i-1);
    endfor;

```

Figure 8-11: *Continued.*

```

    // Set the sentinel value in the temporary array
    // to be equal to the minimum value in the array
    // to be sorted.
    TempArray(0) = GetMinArrayValue(Array);

end;

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Function:    RemoveSentinel
//
// Purpose:     This function adds a sentinel value to an array.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
function VOID RemoveSentinel(TempArray, Array)
INT    iSize, i;
begin

    // Get original array size.
    iSize = SizeOf(Array);

    for i=0 to iSize-1
        Array(i) = TempArray(i+1);
    endfor;

end;

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Function:    ListToArray
//
// Purpose:     This function converts a list to an array.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
function INT ListToArray(List, Array, iType)
INT    iSize, i, iValue;
STRING strValue;
begin

    iSize = ListCount(List);
    if(iSize <= 0) then
        return -1;
    endif;

    // Set array to the same size as the list.
    Resize(Array, iSize);

    ListSetIndex(List, LISTFIRST);

```

Figure 8-11: *Continued.*

```

    if(iType = NUMBERLIST) then
        for i=0 to iSize-1
            ListCurrentItem(List, iValue);
            Array(i) = iValue;
            ListSetIndex(List, LISTNEXT);
        endfor;
    elseif(iType = STRINGLIST) then
        for i=0 to iSize-1
            ListCurrentString(List, strValue);
            Array(i) = strValue;
            ListSetIndex(List, LISTNEXT);
        endfor;
    else
        return -1;
    endif;

    return 0;

end;

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Function:    ArrayToList
//
// Purpose:     This function converts an array to a list.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
function INT ArrayToList(Array, List, iType)
    INT    iSize, i, iValue;
    STRING strValue;
begin

    iSize = SizeOf(Array);
    if(iSize <= 0) then
        return -1;
    endif;

    // Make the first element in the
    // list the current element.
    ListSetIndex(List, LISTFIRST);

    if(iType = NUMBERLIST) then
        for i=0 to iSize-1
            iValue = Array(i);
            ListSetCurrentItem(List, iValue);
            ListSetIndex(List, LISTNEXT);
        endfor;
    elseif(iType = STRINGLIST) then

```

Figure 8-11: *Continued.*


```

        for i=0 to iSize-1
            strValue = Array(i);
            ListSetCurrentString(List, strValue);
            ListSetIndex(List, LISTNEXT);
        endfor;
    else
        return -1;
    endif;

    return 0;

end;

```

Figure 8-11: *Continued.*

This code creates a number of *private* functions that are not exported when you compile the script library. These private functions support the exported functions in performing the sorting activity. One of these private functions is `Swap`, which is used to swap two values in an array. Note that the temporary local variable is of type `VARIANT`. Using a variable of type `VARIANT` permits the swapping of either integer or string values. In order to send an array to a function, the formal parameter needs to be declared as type `VARIANT`. Many sorting algorithms use a sentinel value to avoid a constant check to prevent a loop statement from overrunning the bounds of an array. In the implementation of the shell sort algorithm, the `ShellSort` function uses a sentinel value. There are three private functions devoted to the creation and removal of a sentinel value from the array to be sorted.

In the implementation of the sorting function for a list, the program cheats a bit. Working with lists requires using the list built-in functions. To perform a swap of two elements in a list requires a minimum of eight function calls. This does not count the number of function calls that are required just to perform a comparison between two elements. In the implementation provided for sorting a list using the quicksort algorithm, this program first converts a list into an array, performs the sort on the array, and then converts the array back into a list. The cost to convert a list into an array is linear with the size of the list and going in the other direction is also linear. This is not a high price to pay in order to efficiently sort a list.

COMPILING THE SCRIPT LIBRARY

You need to create a script library from the command line. There are a number of steps that you need to follow in order to create a sorting script library file and use it to

perform the sorting of arrays and lists. These steps are identified below and then discussed in more detail.

1. Add the location of Compile.exe to the PATH environmental variable so you can compile in any location.
2. Modify Sort.rul to be properly set up for compiling it into a script library.
3. Create a command-line file for identifying the location of the header files required for compiling Sort.rul.
4. Compile Sort.rul from the command line to create Sort.obl, which is the script library.
5. Create Sort.h to contain the correct exported function prototypes that you will use in Setup.rul.
6. Copy Sort.obl and Sort.h to a permanent location where your script libraries will be kept.
7. Define the location of the script library and header file in the Compile Folders.ini file.
8. Modify the Setup.rul file to test the sorting functions from the script library.

To accomplish Step 1, you need to add the following location to the path environment variable for the current user.

```
C:\Program Files\InstallShield\Developer\System
```

As an example, you can edit the environment variables on Windows 2000 by doing the following:

1. Access the System Properties dialog from the System applet in the Control Panel or launch it by right clicking on the My Computer icon on the desktop and selecting Properties.
2. In the System Properties dialog click on the Advanced tab and then click the Environmental Variables button.

3. Edit the path environment variable for the current user. You could add this path to the system variables instead of to the current users environment variables if you wanted to make this location available for all users of the machine. Making this change allows you to compile a script from the command line from any location.

Next, you need to make some simple changes to `Sort.rul`. You need to remove the comments from the include statement for the header file `ifx.h` and from the prototype statements placed directly before the definitions of the four sorting functions. The following statements in `Sort.rul` should now be uncommented.

```
#include "ifx.h"
export prototype VOID SelectionSort(BYREF VARIANT);
export prototype VOID ShellSort(BYREF VARIANT);
export prototype VOID QuickSort(BYREF VARIANT, INT, INT);
export prototype VOID ListQSort(BYVAL LIST);
```

The use of the `export` keyword indicates that these four functions are going to be accessed from outside the script library. The other functions in `Sort.rul` are available only inside the script library and you will not be able to call them from `Setup.rul`.

One of the compiler options is to put all command line options inside of a text file so you do not have to type them in on the command line. This option permits a much shorter command line. Figure 8-12 shows what you should put in this text file.

```
"Sort.rul"
"/IC:\Program Files\InstallShield\Developer\Script\Ifx\Include"
"/IC:\Program Files\InstallShield\Developer\Script\iswi\Include"
"/IC:\Program Files\InstallShield\Developer\Script\isrt\Include"
"/IC:\Program Files\InstallShield\Developer\Script\Include"
"/L"
```

Figure 8-12: *The `commandline.txt` file for defining the compiler options.*

You can name this text file anything, such as `commandline.txt`. Place this file in the same location as `Sort.rul`. `Sort.rul` is located in the following folder:

```
C:\MySetups\Learning InstallScript
```

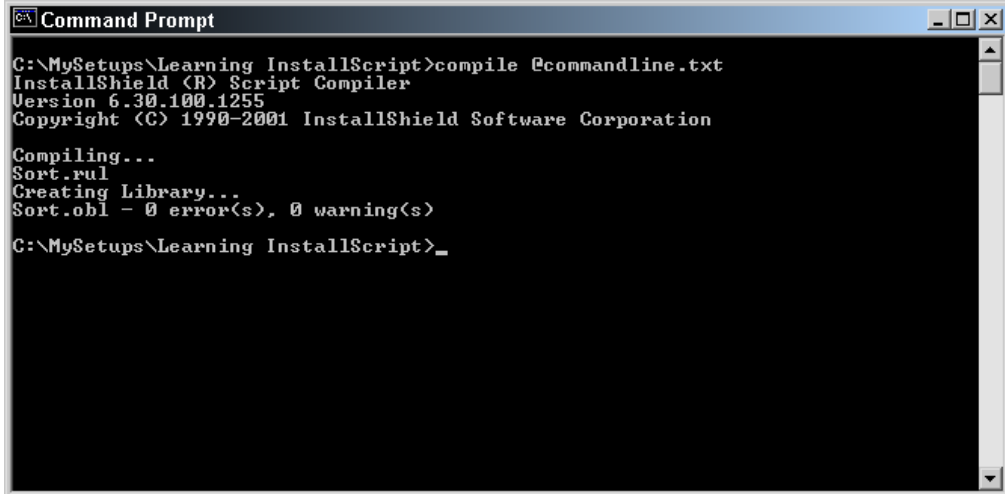
The first line in `commandline.txt` is the name of the file that you are going to compile and this is followed by the four locations where the necessary header files are located. The `/I` switch indicates that the following path is the location of a folder that holds

one or more header files that are needed for the compilation. It is important that the specification of these four locations for finding header files be in the order shown above, otherwise compilation errors will result. The final line in `commandline.txt` is the switch that tells the compiler that you want to create a script library. Note that each item on the command line needs to be on a separate line in `commandline.txt` and that each item including the switch needs to be inside double quotes.

Before compiling `Sort.rul`, make sure that you have saved your changes. You also need to make the current directory the location where `Sort.rul` is located. Now that you have created `commandline.txt`, the command line to compile is as follows:

```
compile @commandline.txt
```

The result of running this command line should be the same as shown in Figure 8-13. Note that when you run the compiler from the command line, the version of the compiler is part of the display (Figure 8-13). The version shown here indicates that this is the same compiler that ships with InstallShield Professional version 6.30. Because the compilers are the same, the InstallScript language information provided in this book is also applicable to InstallShield Professional – Standard Edition.



```
Command Prompt
C:\MySetups\Learning InstallScript>compile @commandline.txt
InstallShield (R) Script Compiler
Version 6.30.100.1255
Copyright (C) 1990-2001 InstallShield Software Corporation

Compiling...
Sort.rul
Creating Library...
Sort.obl - 0 error(s), 0 warning(s)

C:\MySetups\Learning InstallScript>_
```

Figure 8-13: *The command prompt for creating the `sort.obl` script library.*

The next step is to create a header file that will be used in Setup.rul to let the compiler and linker know that the definition of the sorting functions is found in a library file. Name this header file Sort.h. The contents of this file are shown in Figure 8-14.

```

////////////////////////////////////
//
// File Name:      Sort.h
//
// Description:    Header file for the sorting script library
//
// Comments:      This script demonstrates the creation
//                of a script library of sorting functions.
//
////////////////////////////////////

external prototype VOID SelectionSort (BYREF VARIANT);
external prototype VOID ShellSort (BYREF VARIANT);
external prototype VOID QuickSort (BYREF VARIANT, BYVAL INT,
                                   BYVAL INT);
external prototype VOID ListQSort (BYVAL LIST);

```

Figure 8-14: *The sort.h header file for the script library example.*

The next thing that you need to do is create a permanent location for the library file and the header file. A good location is under the folder where all your projects are being created. A suggested folder structure is as shown in Figure 8-15.

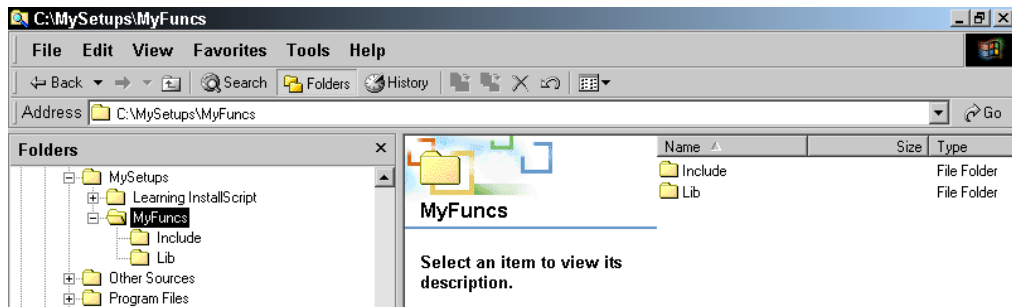


Figure 8-15: *The folder structure for storing our script libraries and associated header files.*

You need to copy Sort.obl to C:\MySetups\MyFuncs\Lib and Sort.h to C:\MySetups\MyFuncs\Include. Before you can easily use the new script library in Setup.rul, you need to identify the location where your library and header

files are located in some global fashion. Do this by editing the `Compile Folders.ini` file, located in the following location:

```
C:\Program Files\InstallShield\Developer\Support
```

There are two sections in this initialization file. The first section identifies the location of header files and the second section names the script libraries that are to be scanned during the linking process. After you edit this file, it should look like what is shown in Figure 8-16. When you make entries in the `Compile Folders.ini` file you are creating header file and library locations that will be used for all projects. For the location of libraries you can also identify project specific locations. The creation of project specific locations for script libraries can be defined by making entries in the `ISLinkerLibrary` table in the Direct Editor. The entries in this table would be the same as you are making in the `Compile Folders.ini` file.

```
[Folders]
Folder0=<ISProductFolder>\Script\ISWi\Include
Folder1=<ISProductFolder>\Script\ISRT\Include
Folder2=<ISProductFolder>\Script\IFX\Include
Folder3=<ISProductFolder>\Script\Include
Folder4=<ISProjectFolder>\MyFuncs\Include

[Libraries]
Libraries1=<ISProjectFolder>\MyFuncs\Lib\Sort.obl
```

Figure 8-16: *The `Compile Folders.ini` file to include the sorting library location information.*

What you have added to this file is the line that locates `Folder4` and the line that locates `Libraries1`. Because you placed your script library and associated header file in the same location where your projects are being built, you can use the predefined path variable `<ISProjectFolder>` to locate the `Include` and `Lib` folders.

All you need to do now is make some minor changes to `Setup.rul` before testing that the sorting script library works as designed. You need to remove the include statement that added the `Sort.rul` file to `Setup.rul`. This statement is at the end of `Setup.rul`. You also need to remove the prototypes for the sorting functions in the script library since the correct prototypes are now in `Sort.h` header file. The last thing that you need to do is add `Sort.h` using the following statement:

```
#include "sort.h"
```

The top of your Setup.rul file should now look like what is shown in Figure 8-17.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script tests the functions to be
//                built into a script library of sorting functions.
//
////////////////////////////////////

#include "ifx.h"
#include "sort.h"

#define CAPTION    "Feedback"

// Prototypes of private functions.
prototype STRING CreateArrayDisplay(BYVAL VARIANT);
prototype STRING CreateListDisplay(BYVAL LIST);

INT      i, iVal, iSizeInt, iSizeStr, iArray(20), iValue;
STRING   strArray(10), strValue;
LIST     iList, strList;

program

```

Figure 8-17: *The revised setup.rul file so as to use the sorting functions from the script library.*

You can now compile Setup.rul and test that your sorting functions still work correctly. You now have a script library that you can use in any installation program where you need to sort arrays or lists.

Adding the Sorting Functions to the Function Wizard

The key to adding a function to the Function Wizard is the file Funcwiz.ini that is found in the same location as the Compile Folders.ini file. The Funcwiz.ini file is composed of three major areas. The first area is a section that defines all the function categories that appear in the Function Wizard. The second area is where, for each category, there is a definition of the functions that are provided in that category. There is one section in the initialization file for each category that is defined in the first section. The third major area is a description of each function defined in the Function Wizard. It is this description that allows the Function Wizard to insert a call to the function in your script.

Before you can use the Function Wizard to add a sorting function to your script, you need to add the functions to Funcwiz.ini. The Function Wizard does not display the list of categories in alphabetical order unless the entries under the first section are made in alphabetical order. However, the list of functions for any category is sorted alphabetically. All you have to do is add your functions at the end of the appropriate sections and they will appear in the Function Wizard in sorted order.

To add your functions to Funcwiz.ini:

1. Give your sorting functions a category of `Sorting` by adding this name to the `[FuncWiz - Category]` section in the Funcwiz.ini file.
2. At the end of the `[FuncWiz - Category - All]` section, add the names of your functions.
3. Create a new category section where your sorting functions will also be listed. Name this new section `[FuncWiz - Category - Sorting]`.
4. Add four sections at the end of Funcwiz.ini to provide the information the Function Wizard needs to insert these functions into the script. A complete new version of Funcwiz.ini is available on the CD-ROM.

Figure 8-18 shows the changes described above.

```
[FuncWiz - Category]
Item1=All
Item2=Batch file
...
...
...

Item20=Sorting
Item21=String
Item22=Version checking
Item23=Uninstallation
Item24=User interface
...
```

Figure 8-18: *The changes that need to be made to the Funcwiz.ini file to add the sorting functions.*


```

[FuncWiz - Category - All]
Item1=AddFolderIcon
Item2=AddProfString
...
...
Item287=ListQSort
Item288=SelectionSort
Item289=QuickSort
Item290=ShellSort

...
...

[FuncWiz - Category - Sorting]
Item1=SelectionSort
Item2=ShellSort
Item3=QuickSort
Item4=ListQSort

...
...

[ListQSort]
SampleLine=ListQSort(List)
Description=Sorts a list using the quicksort algorithm.
Param1Name=List
Param1Desc=The name of the list that is to be sorted.
  [SelectionSort]
SampleLine=SelectionSort(Array)
Description=Sorts an array using the selection sort algorithm.
Param1Name=Array
Param1Desc=The name of the array that is to be sorted.
  [ShellSort]
SampleLine=ShellSort(Array)
Description=Sorts an array using the shellsort algorithm.
Param1Name=Array
Param1Desc=The name of the array that is to be sorted.

[QuickSort]
SampleLine=QuickSort(Array, iLower, iUpper)
Description=Sorts an array using the quicksort algorithm.
Param1Name=Array
Param1Desc=The name of the array that is to be sorted.
Param2Name=iLower
Param2Desc=The name of the lower bound for the array to be sorted.
Param3Name=iUpper
Param3Desc=The name of the upper bound for the array to be sorted.

```

Figure 8-18: *Continued.*

The Funcwiz.ini file handles combo boxes that provide a selection of constants in certain function descriptions. There are two types of combo boxes-- one that cannot be edited and one that can. You can specify a combo box by one the following keyword/value pairs.

```
Param1Type=Combo
Param1Type=EdCombo
```

The Param1Type keyword indicates that the Function Wizard has to create a combo box for the first parameter of the function. The combo box provides a selection of arguments to pass to the function. The number indicates which function parameter is to use a combo box. The value Combo means that the end user cannot add values to what is offered by the combo box. The EdCombo means that the end user can add values in addition to what is provided by the combo box.

Values are defined in Funcwiz.ini by the use of the Param1Val1=value keyword-value pair. For example the SprintfBox function provides three possible values in the combo box for the first argument to the function. These values are defined as follows:

```
Param1Type=Combo
Param1Val1=INFORMATION
Param1Val2=SEVERE
Param1Val3=WARNING
```

This set of keyword-value pairs tells the Function Wizard that there are only three choices provided for the first argument of the SprintfBox function. However, you can use any of the valid message box styles in this location. You could change this combo box to allow additional arguments to be added and displayed in the Function Wizard. After you have inserted the function call into the script, you can change any of the parameters.

Arrays of Structures

InstallScript does not currently allow you to use structures to create a linked list, but you can create an array of structures. The secret to doing this is to use the capability to return a value from a function and, in this fashion, create new memory that you can then assign to an element in an array. The best way to understand this is to look at the example shown in Figure 8-19.

The program in Figure 8-19 contains a function that creates a structure to hold the values that populate a row in the ListBox table based on the arguments passed to the function. This structure is returned to the calling program as a variable of type OBJECT.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script tests the creation of
//                an array of structures.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION      "Feedback"

prototype OBJECT CreateListBoxRow(BYVAL STRING, BYVAL INT,
                                  BYVAL STRING, BYVAL STRING);

// Define a structure that will hold
// a row from the ListBox table.
typedef LBROW
begin
    STRING  strProperty[73];
    INT     iOrder;
    STRING  strValue[65];
    STRING  strText[65];
end;

INT      i;
STRING  szValue, szText;
OBJECT  objListBoxRow(10), obj;
VARIANT Value;

program

    // Create 10 rows for the ListBox table
    // place those rows in an array.
    for i=0 to 9
        Value = i+1;
        szValue = Value;
        szText = "This is row " + Value;
    
```

Figure 8-19: *An example showing the creation of an array of structures.*

```

        set obj = CreateListBoxRow("LISTBOXPROPERTY",
                                   i+1, szValue, szText);
        objListBoxRow(i) = obj;
    endfor;

    // Loop through the array and display
    // the values in each row in the array.
    for i=0 to 9
        SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
                  "ListBox Row %d\n\n%s %d %s %s",
                  i+1, objListBoxRow(i).strProperty,
                  objListBoxRow(i).iOrder,
                  objListBoxRow(i).strValue,
                  objListBoxRow(i).strText);
    endfor;

endprogram

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// Function:   CreateListBoxRow
//
// Purpose:   This creates a structure that contains the values
//            needed to populate the ListBox table.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function OBJECT CreateListBoxRow(strProperty, iOrder,
                                 strValue, strText)

LBROW    lbRow;
begin

    lbRow.strProperty = strProperty;
    lbRow.iOrder      = iOrder;
    lbRow.strValue    = strValue;
    lbRow.strText     = strText;

    // Return an object that is a row
    // in the ListBox table.
    return lbRow;

end;

```

Figure 8-19: *Continued.*

In the calling program, the object returned by the `CreateListBoxRow` function is added as an element in an array of type `OBJECT`. The last part of the calling program uses the structure member operator to display the values in the structure and to display these values to verify that it has created an array of structures. The reason that this approach works is that a structure in `InstallScript` is a COM object. There is

an external reference counting mechanism that keeps each structure alive in memory. Even though you cannot create a true linked list, the ability to create an array of structures can be very useful.

Calling Functions in a DLL

When you call a function in a DLL, the DLL needs to be loaded into memory before the call can succeed. Also you need to know the exact name of the function you are going to call in the DLL. This means that the function needs to be C callable. This means that there is no decoration on the exported name of the function.

There are two main categories of DLL functions that you can call from InstallScript. There are functions in DLLs that you create and functions in Windows DLLs. Functions in Windows DLLs are Windows APIs and it is these functions that are used to create all Windows programs.

User-Defined DLL Functions

There is a special format for prototyping functions that you are going to call from a DLL that you create. This format is shown as follows. Note that the square brackets mean that the item is optional:

```
prototype [calling-convention] [return-type]  
          DLL-name.function-name( [parameter-data-type-list ] );
```

For the calling convention specification there are two keywords that are available: `stdcall` and `cdecl`. The default calling convention is the `stdcall` calling convention. This calling convention is the calling convention used by all Win32 API functions. The `cdecl` calling convention is the default calling convention used by C and C++ programs. The main difference between the two calling conventions is that, when `stdcall` is used, it is the responsibility of the called function to clean up the stack. When the `cdecl` convention is used, the caller of the function has to clean up the stack. There is another difference that is related to name decoration but this is not important since you will create your DLLs with no name decoration. The return type is just as it is for user-defined functions. The name of the DLL and the name of the exported function are delimited by a period. The name of the function has to be

exactly as it is exported by the DLL in the export table. The easiest way to ensure that you know the name of the exported function is to use a module definition file when you compile your DLL.

As an example of calling a function in a DLL, you will create a DLL that validates a serial number that is entered by the user. The script for this example uses an SD dialog that has an edit field for entering a serial number. Look at the code for the DLL function that you are going to call from your script (Figure 8-20). The DLL code that is shown in Figure 8-20 is just for the body of the `ValidateSerialNo` function and the module definition file that forces the exported name of the function to be what you want it to be. In the DLL function you pass the serial number that is entered by the end user, as well as the acceptable values for the first two parts of the serial number.

```
//
// SerialNumber.cpp: Defines the entry point for the DLL application.
//

#include "stdafx.h"

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

//
// A function that validates the entry of a serial number
// made during an application's installation.
//
STDAPI ValidateSerialNo(LPSTR SerialNumber, LPCSTR Product,
                       LPCSTR Version, DWORD SeqStart,
                       DWORD SeqInc)
{
    TCHAR delimiters[] = " -";
    TCHAR *tokens[4];
    DWORD SeqNo;
    int i=0;
```

Figure 8-20: *The code for the `ValidateSerialNo` DLL function.*

```

// Obtain all the tokens in a serial number
// and assign them to an array.
tokens[i] = _tcstok(SerialNumber, delimiters);
while(tokens[i]!=NULL && i<3)
{
    i++;
    tokens[i] = _tcstok(NULL, delimiters);
};

// Make sure that only the correct number of tokens was used.
if((tokens[i]!=NULL && i>=3) || tokens[i]==NULL && i<3)
    return -1;

// Make sure that the sequence number contained only numbers.
if((SeqNo = atol(tokens[2])) == 0)
    return -2;

// Check for a valid product code.
if(_tcsicmp(tokens[0], Product) != 0)
    return -3;

// Check for a valid version number.
if(_tcsicmp(tokens[1], Version) != 0)
    return -4;

// Validate that the sequence number is large enough.
if(SeqNo < SeqStart)
    return -5;

// Validate that the sequence number is in a valid sequence.
if(((SeqNo - SeqStart) % SeqInc) != 0)
    return -6;

// Return success if we get this far.
return 0;
}

;
; SerialNumber.def
;
; Module definition file for the SerialNumber.dll
;
LIBRARY    "SERIALNUMBER"

DESCRIPTION "DLL for validating a serial number"

EXPORTS
    ValidateSerialNo    PRIVATE

```

Figure 8-20: *Continued.*

The DLL tokenizes the entered serial number so that each separate part can be validated separately. As constructed the `ValidateSerialNo` function is set up to check a serial number that has three parts. The first part is the product identifier, the second part is the version number, and the third part is a sequence number that is different for each user.

The validation for the first two parts is a simple comparison between what the user enters and the values that the script passes to the DLL. The validation of the sequence number ensures that the sequence number is a whole multiple of a specified increment plus a specified starting number. The program checks this by subtracting the starting number from the sequence number and then checking if the modulus of this number by the increment is zero. If it is, then the sequence number that was entered is valid.

Now that you have looked at the DLL function code, you need to look at the script that will call this function (Figure 8-21).

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script tests the use of
//                functions in a dynamic link library.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION      "Feedback"
#define MAX_TRYES   3

prototype stdcall LONG SerialNumber.ValidateSerialNo(BYVAL STRING,
              BYVAL STRING, BYVAL STRING, BYVAL LONG, BYVAL LONG);

STRING  szTitle, szMsg, svName, svCompany, svSerial;
STRING  szProduct, szVersion, szDLLName;
INT     iReturn, iTryCnt;
LONG    lValidate, lStart, lIncrement;

program

```

Figure 8-21: *Setup.rul* that demonstrates the calling of a function in a DLL.


```

// Set the location of the DLL.
szDLLName = SUPPORTDIR ^ "SerialNumber.dll";
UseDLL(szDLLName);

// Initialize the arguments to be used in the
// SD dialog and the function call to the DLL.
szTitle = "Customer Information";
szMsg = "Please enter the requested information.";
szProduct = "ABCDEF";
szVersion = "0750";
lStart = 1000000519;
lIncrement = 519;
iTryCnt = 1;

RegisterUser:

// Null the serial number every time we
// enter the SdRegisterUserEx dialog box.
svSerial = "";

// Only let a certain number of attempts at
// entering a serial number.
if(iTryCnt > MAX_TRY) then
    MessageBox("Too many attempts.\n" +
               "The installation will now terminate.", SEVERE);
    abort;
endif;

// Display the SdRegisterUserEx dialog box.
iReturn = SdRegisterUserEx(szTitle, szMsg, svName,
                           svCompany, svSerial);

// If the user clicks the Back button
// then come back into the dialog.
if(iReturn = BACK) goto RegisterUser;

// When the Next button is clicked validate the
// serial number that was entered.
if(iReturn = NEXT) then
    lValidate = ValidateSerialNo(svSerial, szProduct,
                                szVersion, lStart, lIncrement);
endif;

// Display the result as returned from the DLL function.
switch(lValidate)
case 0:
    sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
              "Validation was successful");

```

Figure 8-21: *Continued.*

```

case -1:
    sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
               "Bad serial number was entered");
    iTryCnt++;
    goto RegisterUser;
case -2:
    sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
               "Sequence number is not a number.");
    iTryCnt++;
    goto RegisterUser;
case -3:
    sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
               "Product entry not valid.");
    iTryCnt++;
    goto RegisterUser;
case -4:
    sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
               "Version entry not valid.");
    iTryCnt++;
    goto RegisterUser;
case -5:
    sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
               "Sequence number too small.");
    iTryCnt++;
    goto RegisterUser;
case -6:
    sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
               "Invalid sequence number.");
    iTryCnt++;
    goto RegisterUser;
default:
    sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
               "Unknown validation error");
    iTryCnt++;
    goto RegisterUser;
endswitch;

// Free the DLL from memory.
UnUseDLL("SerialNumber.dll");

endprogram

```

Figure 8-21: *Continued.*

The code shown in Figure 8-21 gives the end user three attempts to enter the correct serial number and then aborts the installation. Note that in the prototype for the DLL function, all arguments are passed by value because no values are returned through the arguments. The only return is through the return statement in the DLL. The

value that is returned to the calling script identifies what was wrong with the input made by the end user.

To use a DLL that you create, you need to specifically load it into memory. When you are finished with it, you need to free the memory. You load a DLL into memory by using the `UseDLL` function and free the memory by using the `UnUseDLL` function. For the `UseDLL` function, you need to provide the complete path to the DLL. This brings up the question of how and where to copy the DLL to the target system so that it can be used during the installation but removed when the installation is complete.

First, you need to place the `SerialNumber.dll` file in a central location so all projects can easily get to it. Such a location is as follows:

```
C:\MySetups\Setup Files
```

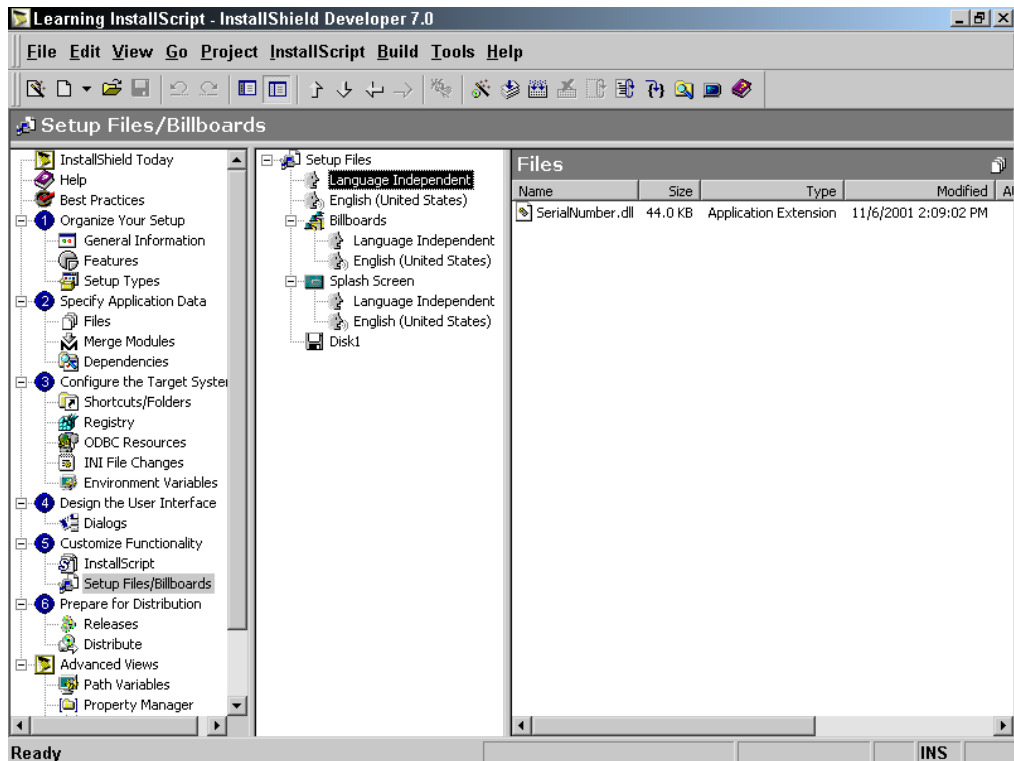


Figure 8-22: *The Setup Files/Billboards view and the insertion of a DLL into the setup.*

To include `SerialNumber.dll` in the installation program as shown in Figure 8-22 do the following:

1. Navigate to the Setup Files/Billboards view and click on the Language Independent node. It is in the Setup Files/Billboards view that you include those files that are needed just for the installation.
2. Right-click in the Files pane and select Insert Files.
3. Browse to where the file `SerialNumber.dll` is located and click Open. This inserts the file into the installation project. This file will be copied to a location defined by the `SUPPORTDIR` system variable when the installation is run.

In `Setup.rul`, you need to create a `STRING` variable that holds the path to `SerialNumber.dll` during the installation. This line of code is formed from the concatenation of `SUPPORTDIR` and the name of the DLL as follows:

```
szDLLName = SUPPORTDIR ^ "SerialNumber.dll";
```

This statement uses the special path concatenation operator that inserts the backslash into the string if it is needed.

Calling Windows APIs

There is very little difference between using a Windows API function and a user-defined function in a DLL. You have to prototype a Windows API function, but you do not have to load the Windows DLL into memory because it was loaded when Windows booted. The trick to calling a Windows API function is to know which DLL exports what function. The best way to find out which Windows DLL to use in the prototype is to go to the MSDN library and look up the function that is going to be used. This tells which DLL exports the function. The format for prototyping a Windows API function is as follows:

```
prototype [calling-convention] [return-type]  
    KERNEL32|USER32|GDI32.API-name( [parameter-data-type-list ] );
```

The calling convention can be used in the prototype but it is not necessary for a Windows API function since the default calling convention is what is used by Win32

APIs. If you do not specify a return type, the default expected by InstallScript is an integer return type.

In the above prototype format there are the names of three Windows DLLs, one of which you need to use. There are other Windows DLLs but these are the ones that export the more commonly used functions.

The example shown in Figure 8-23 demonstrates how calling a Windows API works. This example uses a Windows API function to get the system time and then uses another function to convert the system time to local time.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script tests the calling of
//                Windows API functions.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION    "Feedback"

// Prototype the two Windows APIs to be used in the program.
prototype stdcall VOID KERNEL32.GetSystemTime(POINTER);
prototype stdcall BOOL KERNEL32.SystemTimeToTzSpecificLocalTime
                    (POINTER, POINTER, POINTER);

// Define a structure for obtaining
// the system time.
typedef _SYSTEMTIME
begin
    SHORT    iYear;
    SHORT    iMonth;
    SHORT    iDayOfWeek;
    SHORT    iDay;
    SHORT    iHour;
    SHORT    iMinute;
    SHORT    iSecond;
    SHORT    iMilliseconds;
end;

```

Figure 8-23: *Setup.rul* that shows the calling of several Windows API functions.

```

_SYSTEMTIME      SystemTime, LocalTime;
_SYSTEMTIME POINTER pSystemTime, pLocalTime;
VARIANT Day, Year, Hour, Minute, Second;
STRING  szDisplay, szMonth(12), szDayOfWeek(7);
BOOL    bSuccess;

program
    // Initialize an array of month names.
    szMonth(0) = "January"; szMonth(1) = "February";
    szMonth(2) = "March"; szMonth(3) = "April";
    szMonth(4) = "May"; szMonth(5) = "June";
    szMonth(6) = "July"; szMonth(7) = "August";
    szMonth(8) = "September"; szMonth(9) = "October";
    szMonth(10) = "November"; szMonth(11) = "December";

    // Initialize an array of day of week names.
    szDayOfWeek(0) = "Sunday"; szDayOfWeek(1) = "Monday";
    szDayOfWeek(2) = "Tuesday"; szDayOfWeek(3) = "Wednesday";
    szDayOfWeek(4) = "Thursday"; szDayOfWeek(5) = "Friday";
    szDayOfWeek(6) = "Saturday";

    // Get pointers to the two _SYSTEMTIME structures.
    pSystemTime = &SystemTime;
    pLocalTime = &LocalTime;

    // Call the Windows API to get the system time.
    GetSystemTime(pSystemTime);

    // Convert the system time to a display string.
    szDisplay = "System Time\n\nDate:\t";
    Year = pSystemTime->iYear;
    Day = pSystemTime->iDay;
    szDisplay = szDisplay + szDayOfWeek(pSystemTime->iDayOfWeek) +
        ", " + szMonth(pSystemTime->iMonth-1) + " " + Day +
        ", " + Year + "\nTime:\t";

    Minute = pSystemTime->iMinute;
    Second = pSystemTime->iSecond;
    if(pSystemTime->iHour > 12) then
        Hour = pSystemTime->iHour - 12;
        szDisplay = szDisplay + Hour + ":" + Minute + ":"
            + Second + " PM";
    else
        Hour = pSystemTime->iHour;
        szDisplay = szDisplay + Hour + ":" + Minute + ":"
            + Second + " AM";
    endif;

```

Figure 8-23: *Continued.*

```

// Display the system time.
sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION, "%s", szDisplay);

// Call the Windows API to convert from
// system time to local time.
bSuccess = SystemTimeToTzSpecificLocalTime(NULL,
                                             pSystemTime, pLocalTime);

// Check to see if the conversion from
// system time to local time succeeded
// before converting the local time
// to a display string.
if(bSuccess) then
    szDisplay = "Local Time\n\nDate:\t";
    Year = pLocalTime->iYear;
    Day = pLocalTime->iDay;
    szDisplay = szDisplay +
        szDayOfWeek(pLocalTime->iDayOfWeek) +
        ", " + szMonth(pLocalTime->iMonth-1) + " " + Day +
        ", " + Year + "\nTime:\t";

    Minute = pLocalTime->iMinute;
    Second = pLocalTime->iSecond;

    if(pLocalTime->iHour > 12) then
        Hour = pLocalTime->iHour - 12;
        szDisplay = szDisplay + Hour + ":" + Minute + ":"
            + Second + " PM";
    else
        Hour = pLocalTime->iHour;
        szDisplay = szDisplay + Hour + ":" + Minute + ":"
            + Second + " AM";
    endif;

    // Display the local time.
    sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION, "%s",
              szDisplay);
else
    // Display that there was a conversion error.
    sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
              "Unable to convert from system time to local time." );
endif;

endprogram

```

Figure 8-23: *Continued.*

In this example, the `GetSystemTime` windows API function retrieves the system time. The argument to this function is a pointer to a `SYSTEMTIME` structure, which

is populated by the function. System time is in Coordinated Universal Time (UTC). This means that system time is the same as Greenwich Mean Time. To get the local time you need to perform a conversion from UTC to local time. The program converts the system time using the `SystemTimeToTzSpecificLocalTime` function.

First, the example program gets the system time and then converts values into a display string. It then converts this time to local time and displays it again. For the conversion process, the program passes as an argument the system time structure that was populated by the call to the `GetSystemTime` function and a pointer to an empty structure that is populated with the local time. Both structures are a `SYSTEMTIME` structure.

As a final note there are a number of Windows API functions prototyped in a header file called `WinApi.h`. This file is located in the same location as `Isrt.h`, but `Isrt.h` does not include `WinApi.h`. To make use of these prototypes for many of the common Windows API functions, all you have to do is include `WinApi.h` at the top of your script.

Passing an Array to a DLL Function

The key to understanding how to pass an array to a DLL function is the fact that, in `InstallScript`, an array is handled as a variable of type `VARIANT`. In C/C++, a `VARIANT` is a structure that is a container for a very large union that carries many different data types. One of these data types is something called a `SAFEARRAY`, which is another structure. One of the members in a `SAFEARRAY` structure is the array itself. All the other members in the `SAFEARRAY` structure serve to keep track of the dimensions and bounds of the array. To get a pointer to an array that you can then pass to a DLL function means that you start with a `VARIANT` structure and get a pointer to the `SAFEARRAY` structure and then use this pointer to get to the array. This process is shown in Figure 8-24.

Don't worry if this looks confusing the first time you read through it. Just spend the time to study it and you will get the idea. This is a perfect example of why it is important to understand how the `InstallScript` engine works.


```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:       This script demonstrates how to pass
//                 an array to a DLL function.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION      "Feedback"
prototype stdcall BOOL PSAPI.EnumProcesses(BYVAL POINTER,
                                             BYVAL LONG, BYREF LONG);

// Duplicate the C/C++ definition
// of the SAFEARRAY structure.
typedef cSAFEARRAY
begin
    SHORT    cDims;
    SHORT    fFeatures;
    LONG     cbElements;
    LONG     cLocks;
    POINTER  pvData;
    /* ignore the SAFEARRAYBOUND element */
end;

// Duplicate the C/C++ definition
// of the VARIANT structure.
typedef cVARIANT
begin
    SHORT    vt;
    SHORT    wReserved1;
    SHORT    wReserved2;
    SHORT    wReserved3;
    POINTER  pData;
end;

INT        i, iSize;
STRING     szDisplay;
HWND       ProcHandle;
VARIANT    Value;
BOOL       bReturn;
LONG       idProcesses(1024), cb, cbNeeded;
cVARIANT   POINTER pVariant;
cSAFEARRAY POINTER pArray;

```

Figure 8-24: *Setup.rul* that shows how to pass an array to a DLL function.

```

program

    // Get a pointer to the array
    // which is actually a pointer
    // a cVARIANT structure.
    pVariant = &idProcesses;

    // Since the data in the cVARIANT structure
    // is a structure of type cSAFEARRAY we get
    // a pointer to the cSAFEARRAY structure.
    pArray = pVariant->pData;
    // Get the size in bytes of the
    // array that will be passed.
    cb = SizeOf(idProcesses) * 4;

    // Using the pointer to the cSAFEARRAY structure
    // we can actually get a pointer to the array itself.
    bReturn = EnumProcesses(pArray->pvData, cb, cbNeeded);

    // Check to see if the function succeeded
    // and resize the array to that which
    // just contains the number of process IDs
    // returned from the function.
    if(bReturn) then
        Resize(idProcesses, cbNeeded/4);
    else
        sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
                  "Function did not succeed.");
        abort;
    endif;

    // Create a display string for the process IDs.
    for i=0 to SizeOf(idProcesses)-1
        Value = idProcesses(i);
        szDisplay = szDisplay + Value + ", ";
    endfor;

    sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
              "Process IDs\n\n%s", szDisplay);

endprogram

```

Figure 8-24: *Continued.*

This example uses the `EnumProcesses` Windows API function because it requires the passing of an array into which the function will insert the IDs of all the processes running on the system. This function is in the `psapi.dll`, which is also loaded into memory when Windows boots. This function takes three arguments: a pointer to

an array, the size of the array, and an argument that will hold the number of bytes that are returned in the array.

Since the program has to dig into the `VARIANT` and `SAFEARRAY` structures to get to the array pointer it passes to the DLL function, it needs to duplicate (where necessary) both the `VARIANT` and `SAFEARRAY` structures in the script.

Getting the pointer to the array is a three-step process:

1. Declare a pointer to the `cVARIANT` structure and then assign it a value equal to the pointer to the array declared.
2. Declare a pointer to the `cSAFEARRAY` structure and set its value to the value in the `cVARIANT` structure that points at the `SAFEARRAY` structure in the `VARIANT` structure.
3. Take this pointer and use it to access the array that is an element of the `SAFEARRAY` structure.

The best way to understand this is to study the code in Figure 8-24.

Passing Strings to Functions

As discussed in Chapter 6, strings in `InstallScript` are auto-sized to hold the text. When you add more text to a string variable, it is auto-sized again. When you pass a string by reference to a user-defined function, the string is auto-sized to the length required. You can define a size for a string when you declare it and, if a greater size is required, it is auto-sized upward. When passed to a user-defined function, the declared size is carried over to the function.

The situation is different when you pass a string to a DLL function. Here strings are not auto-sized by the DLL function, so you need to define a size for a string that is passed to a DLL function by reference. A string being passed by value is in essence a constant and must already have a size since it will have been assigned a value before it is passed. `InstallScript` sizes a string being passed by reference to a DLL function to 1024 bytes. If you size it higher, it will be passed with the larger size value but it will never be passed with a size smaller than 1024 bytes.

When passing strings to a Windows API function, you should ensure that the correct function name is used. Most Windows API functions that are passed strings have two versions, one the ANSI version and one the Unicode version. To indicate which one is which, the ANSI versions of functions have an 'A' appended to the function name and the Unicode versions have a 'W' appended to the function name. When InstallScript passes a string to a DLL function, it converts the string to an ANSI string. This means that, from InstallScript, you can call only the ANSI versions of Windows API functions. Later versions of InstallShield Developer will have the capability to pass either ANSI or Unicode strings to a Windows API. When this change is made you will have access to those Windows APIs that only accept Unicode strings.

Conclusion

There are four categories of functions covered in this chapter. The built-in functions are those that are directly supported by InstallScript without requiring prototyping. The built-in functions can be easily inserted into your scripts using the Function Wizard. The on-line Language Reference provides a complete description of how to use each of the built-in functions. If you want help on a function that is in your script, place the cursor on the function name and press F1. This launches the help for that function.

For the event-handler functions, this chapter discusses the various function categories and discusses each of the supported event-handler functions briefly. The details of how the event-driven architecture works is discussed in detail in Chapter 4. The main focus of this chapter is the creation of user-defined functions. Here you learned about the two types of user-defined functions, entry-point functions and generic functions. The discussion of entry-point functions is covered in Chapter 11, which covers the subject of InstallScript custom actions.

For generic user-defined functions, this chapter provides examples of how to create useful functionality. You also saw how to create your own script libraries and how to insert your script library functions into the Function Wizard. You created a small library of sorting functions, added them to a script library, and then made these functions available through the Function Wizard.

Finally, this chapter looked at the creation and use of functions in a dynamic link library. You saw how to prototype these functions and to use functions in DLLs that you create or Windows API functions. You also worked through an example of how to pass an array to a DLL function. This required a basic understanding of both the `VARIANT` and `SAFEARRAY` structures.

Exception Handling and COM

This chapter covers two related concepts: exception handling and the use of COM. These capabilities in InstallScript give you significant power for creating powerful and robust installation programs. Exception handling allows you to manage run-time errors in a fashion that does not leave the end user with a failed installation and no knowledge of what happened. With exception handling, your installation program can invoke a special error handling function or group of statements that are called when an error occurs. It is even possible, depending on the error, to programmatically handle the error without having to terminate the installation.

Being able to access COM objects from your script provides a way to extend the built-in functionality of InstallScript. There are a number of important COM objects that are available on the Windows operating system and you can access these COM objects from your script. This chapter examines three of the most important objects that you can create. We begin our discussion with a look at exception handling since it is important to use exception handling when creating COM objects in InstallScript.

Exception Handling Basics

An exception is something abnormal that occurs in a program. Exception handling is the mechanism that allows two functions to communicate with each other when something abnormal happens. One function calls another function and the called function detects an error and, not knowing how to handle the error, throws an exception so that the calling function can decide how to handle the error. The traditional approach to handling an error has been for the called function to return an error code to the calling function. Then, based on a check of the returned value by the calling function, some action is taken. The concept of exception handling discussed in this chapter is the modern approach to handling an error when it occurs in a program.

It is strongly recommended that you use the exception handling capabilities that are described in this section when you create installation programs. If you do not include exception handling in your programs, the program will terminate when an exception occurs without providing information for the end user. This is not an acceptable approach to creating installation programs. With proper implementation of the exception handling capabilities in InstallScript, many errors can be fixed or at least the program can be implemented to fail gracefully when it encounters an exception.

In InstallScript the exception handling functionality revolves around the `try...catch...endcatch` statement and the `Err` object. Any script code for which you want to monitor an exception must be executed from within a `try` block. If an exception occurs in the code, the statements placed in the `catch` block are executed. Code that throws exceptions outside of a `try` block are not handled using InstallScript's exception handling mechanism.

The `try...catch...endcatch` Statement

The general format of the `try...catch...endcatch` statement is as follows:

```
try // try block
    // Code that is to be monitored for exceptions.
catch // catch block
    // Code that is designed to handle any exception
    // generated in the try block.
endcatch;
```

In the `try` block an exception normally is thrown when you call a function that detects an error. To ensure backward compatibility, InstallScript's built-in functions do not throw exceptions. Because of this, the function that detects an error must be a user-defined function. Built-in functions use the traditional approach and return error codes to the caller of the function. Once an exception is thrown, no further statements in the `try` block are executed and program execution jumps to the first statement in the `catch` block.

The `try...catch...endcatch` statement can be nested inside another `try...catch...endcatch` statement. The general format of nesting is as follows:

```
try // Outer try block
  // Outer code that is to be monitored for exceptions.
  try // Inner try block
    // Code that is to be monitored for exceptions.
    catch // Inner catch block
      // Code that is designed to handle any exception
      // generated in the try block.
    endcatch;
catch // Outer catch block
  // Code that is designed to handle any exception
  // generated in the outer try block.
  try // Inner try block
    // Code that is to be monitored for exceptions.
    catch // Inner catch block
      // Code that is designed to handle any exception
      // generated in the inner try block.
    endcatch; // Inner endcatch
endcatch; // Outer endcatch
```

To understand how to throw an exception you need to understand the `Err` object and its properties and methods.

The Err Object

InstallScript instantiates a built-in object called the `Err` object. This object has global scope and it has six properties and two methods. You do not have to create an instance of this object because it is created when the InstallScript engine starts. You can use the `Err` object to capture exceptions that are thrown by the InstallScript

engine or to capture exceptions that you throw in your user-defined functions. In InstallScript, to retrieve the value of one of the properties, use the following format:

```
Variable = Err.property-name
```

When you want to use one of the two methods, the format is as follows:

```
Err.method-name [ (arguments) ]
```

The properties of the Err object are described below:

Number: This is a read/write property that returns or sets a numeric value specifying an error. This is the Err object's default property.

Description: This is a read/write property that returns or sets a descriptive string that is associated with a particular error. The Description property is a short description of an error that alerts the user about an error that cannot be handled by the installation program.

Source: This is a read/write property that returns or sets the name of the object or application that originally generated the error. The Source property should specify a string expression that is usually the class name or programmatic ID of the object that caused the error. Like the Description property, this information is provided when the installation program is not able to handle the error.

HelpFile: This is read/write property that returns or sets a fully qualified path to a help file. If this property has a value, it is automatically called when the user clicks the Help button or presses the F1 key when in the error message dialog box. The help file can be either a .hlp or a .chm file.

HelpContext: This is a read/write property that returns or sets a context ID for a topic in the help file that is specified in the HelpFile property. If just the HelpFile property is set, the help file is displayed but the end user needs to search for the help topic applicable to the error. If the HelpContext property is set, the help file opens to the relevant help topic for the error.

LastDllError: This is a read-only property that holds the return value of the Win32 API function GetLastError. You cannot just use the GetLastError function to get the error that occurred from calling a DLL function because the InstallScript engine continually calls DLL functions and

result would not be accurate. Using the `LastDllError` property is the correct method to get the value of the error code returned from a DLL function call.

The `Err` object has two methods. One of these methods is used to throw an exception and the other method is used to clear the last error that was thrown. These two methods are discussed in the following list.

Raise: The `Raise` method is used to generate a run-time exception. This method takes up to five arguments in the following format:

```
Err.Raise[(number, source, description, helpfile, helpcontext)];
```

All of these arguments are optional and they are the same as the first five properties described above. You can set the properties first and then call the `Raise` method without any arguments or you can define the value of the properties when you provide them as the arguments of this method. When you are not specifying any arguments, you do not have to include the parentheses with a call to the `Raise` method. However, when arguments are defined, the parentheses are required.

Clear: The `Clear` method takes no arguments. The purpose of this method is to clear the value of all properties and it is used between calls to the `Raise` method.

There is normally a chain of function calls where one function calls another function and that function calls another function. How exceptions are handled in this type of hierarchy is discussed in the next section.

Exception Handling Hierarchy

In normal circumstances an exception that is generated in a function cannot be handled correctly by the function itself. The function that called the function normally has the responsibility to handle the exception. If it cannot handle the exception, it can pass the exception up the chain of function calls if it is not already at the top of the hierarchy. Figure 9-1 provides a diagram of the normal hierarchy of functions and how exceptions are passed up the chain of function calls.

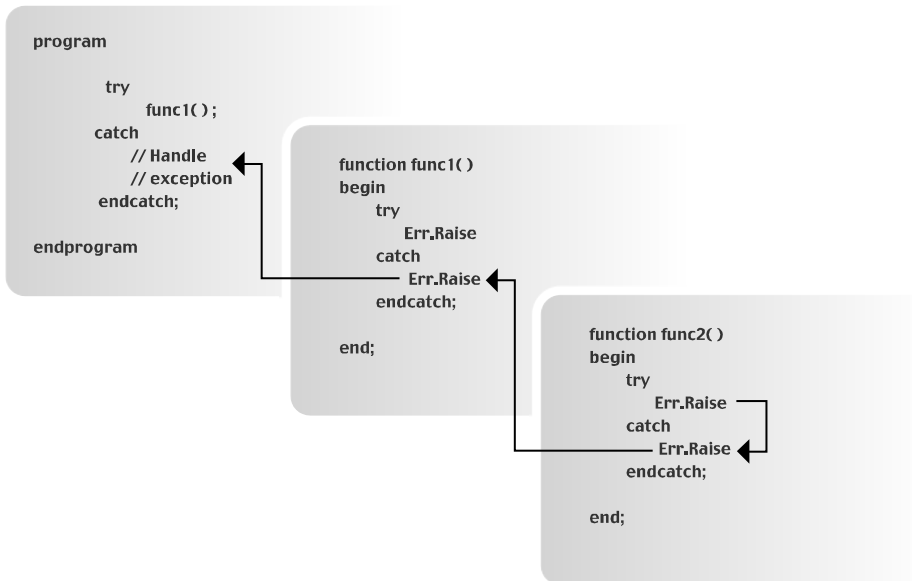


Figure 9-1: *A typical hierarchy of function calls.*

In this figure, the main program calls `func1` and `func1` calls `func2`. If `func2` generates an exception, the exception can be thrown back up to the top of the hierarchy if that is what is necessary. If `func1` can handle the exception then it should do so and not throw the exception any higher in the chain of function calls.

The example program shown in Figure 9-2 demonstrates how the hierarchy of thrown exceptions works. This program shows the exact situation as outlined in Figure 9-1. An exception is generated at the bottom of the calling chain of functions and the exception is raised back up to the top of the calling chain. In a real world installation, the program would handle the exception at the most appropriate location and thus provide the user with a seemingly faultless install experience.

The contrived code shown in Figure 9-2 has a chain of calls from the main program down through three functions. In the last function, called `function3`, the program attempts to access an array with a negative array index, which generates an exception. Program execution jumps immediately to the `catch` block where a message box is displayed and then the `Raise` method is executed.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates the
//                hierarchy of raising an exception.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION    "Feedback"

prototype function1(INT, INT);
prototype function2(INT, INT);
prototype function3(INT, INT);

INT      a, b, i;

program

    a = 1;
    b = 2;

    for i=0 to 1
        try
            function1(a, b);
        catch
            SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
                "The main program can handle the error.\n" +
                "Error number: %d\nError description: %s",
                Err.Number, Err.Description);

            a = 2;
            b = 1;

        endcatch;
    endfor;

endprogram

// function1
function function1(j, k)
begin

```

Figure 9-2: *Setup.rul showing the hierarchy of thrown exceptions.*

```

    try
        function2(j, k);
    catch
        sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
            "function1 cannot handle the error.");
        Err.Raise;
    endcatch;

end;

// function2
function function2(j, k)
begin

    try
        function3(j, k);
    catch
        sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
            "function2 cannot handle the error.");
        Err.Raise;
    endcatch;

end;

// function3
function function3(j, k)
INT    iArray(10), iVal;
begin

    try
        // Generate an exception by accessing an
        // array using a negative array index.
        iVal = iArray(j-k);
        sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
            "The second try works now.");
    catch
        sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
            "function3 cannot handle the error.");
        Err.Raise;
    endcatch;

end;

```

Figure 9-2: *Continued.*

Raising the exception returns control to the catch block in function2, which in turn raises the error again to the catch block of function1. Finally the exception is raised to the top of the calling hierarchy in the catch block of the main program. Here the two values that originally caused the negative array index are

reversed and the program attempts to make the function calls. The second time the array index is positive, so no exception is thrown in `function3`. Now the `SprintfBox` function call that occurs right after the array access is executed and the success message is displayed.

From this example it is evident that the InstallScript engine raises some exceptions. The next section looks at these exceptions.

InstallScript Engine Exceptions

The InstallScript engine throws seven different exceptions (Table 9-1). In Table 9-1 the number in parentheses shown in the Error Code column is the decimal version of the error code.

Table 9-1: The InstallScript Engine Exceptions

Error Code	Error Description
0x80040701 (-2147219711)	This error code indicates a division by zero.
0x80040702 (-2147219710)	This error code indicates that an attempt to load a DLL into memory failed. A DLL is loaded into memory from a script through the use of the <code>UseDLL</code> function. This can occur when either the DLL that is the target of the <code>UseDLL</code> function cannot be found or the DLLs on which the DLL being loaded depend cannot be found.
0x80040703 (-2147219709)	This error occurs when the function that is being called from a DLL cannot be found in the DLL's export table. This can occur when the exported name of the function is not known. It is always best to use a module definition file to force the exported function name to be undecorated. An example of a module definition file is shown in Chapter 8.

Table 9-1: The InstallScript Engine Exceptions (Continued)

Error Code	Error Description
0x80040704 (-2147219708)	This error occurs when a call into a DLL function results in a bad stack. This type of error can occur when a DLL function as prototyped in the script is not the same as the prototype of the function in the DLL. See Chapter 8 about the <code>stdcall</code> and <code>cdecl</code> keywords.
0x80040705 (-2147219707)	This error occurs when a string is accessed outside of its boundaries with one exception. Because of some backward compatibility issues, no string--regardless how short--will have an exception thrown unless a program tries to access location 300 or greater. An attempt to use a negative index triggers this exception.
0x80040706 (-2147219706)	This exception is thrown when a program tries to use an invalid object. Because of this, you should always check the objects that you create using the <code>IsObject</code> built-in function before attempting to use the object.
0x80040707 (-2147219705)	This exception is thrown when a DLL function that you are using crashes. This can happen when the DLL function causes a memory fault.

In Figure 9-3 is shown an example of code that demonstrates several of the InstallScript engine exceptions. This code example uses a switch statement to catch the error and to display a description of the error.

This code example loops through the four statements that it is using to generate the InstallScript engine exceptions. To generate the other exceptions, you would need to create a DLL that you could try and call in various ways. As an example you could create a DLL that has a decorated name for an exported function but still try to call the function using the undecorated name.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates several
//                of the InstallScript engine exceptions.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION    "Feedback"

INT      iArray(10), iVal1, iVal2, iVal3, i;
STRING  szString[10], szVersion;
CHAR    cChr;
OBJECT  objInstaller;

program

    iVal2 = 10;
    iVal3 = 0;

    for i=0 to 3

        try

            switch(i)
                case 0:
                    // This generates error 0x80040701.
                    iVal1 = iVal2/iVal3;
                case 1:
                    // This generates error 0x80040705.
                    cChr = szString[-1];
                case 2:
                    // This generates error 0x80040706.
                    set objInstaller =
                        CreateObject("WindowInstaller.Installer");
                    szVersion = objInstaller.Version;
                case 3:
                    // This generates an unknown error.
                    iVal1 = iArray(-1);
            endswitch;

        catch

```

Figure 9-3: *Setup.rul* that demonstrates some of the InstallScript engine exceptions.


```

switch(Err.Number ^ 0x80040000)
  case 0x701:
    SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
      "There was a division by zero.");
  case 0x702:
    SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
      "DLL failed to load.");
  case 0x703:
    SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
      "DLL function cannot be found.");
  case 0x704:
    SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
      "Bad DLL function prototype.");
  case 0x705:
    SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
      "String accessed outside bounds.");
  case 0x706:
    SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
      "Tried to use an invalid object.");
  case 0x707:
    SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
      "DLL function crashed.");
  default:
    SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
      "Unknown error was thrown\n\n" +
      "Error number: %x\n" +
      "Error description: %s",
      Err.Number ^ 0x80040000,
      Err.Description);
endswitch;
endcatch;
endfor;

SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
  "The error test is complete.");

endprogram

```

Figure 9-3: *Continued.*

In this chapter's section on COM, you will have the opportunity to use InstallScript's exception handling capability. As already mentioned the built-in functions do not throw exceptions. This is because of the need to maintain backward compatibility with scripts created prior to the release of InstallShield Professional – Standard Edition version 6.0. If necessary, however, you could provide wrappers around the built-in functions that threw exceptions based on the return value received from the built-in function.

Creating COM Objects

The InstallScript engine is considered an ActiveX or Automation client through the use of the OBJECT data type discussed in Chapter 6. InstallScript allows you to access any of the functionality that an ActiveX object exposes through the IDispatch interface. An ActiveX object is an instance of a class that can expose three types of members: methods, properties, and events. An ActiveX object is a COM object. A brief description of these three types of members is provided in the following list:

Methods: Methods are actions that an ActiveX object can perform. You can think of methods as functions that you can call from InstallScript to perform activities required during an installation.

Properties: Properties can also be thought of as functions, but these functions retrieve information about the state of an object that you have created. You can access a property to create another object, which will also expose methods, properties, and events.

Events: An event is an action that is recognized by an ActiveX object and which you write code to handle. You can think of events as methods that you write and which the ActiveX calls in response to an event such as a mouse click.

An application that exposes its functionality such that other applications can access it by creating objects is called an ActiveX component. This chapter looks at three of the applications that provide significant functionality for use during an installation program. These three ActiveX components are the Windows Installer, the Scripting run time, and the Windows Scripting Host.

In InstallScript, you will use the `CreateObject` function to create the main object for each of these three ActiveX components. The argument to this function is the COM ProgId that has been registered in the registry. A typical statement that is used in InstallScript to create an ActiveX object is as follows:

```
set object = CreateObject("WindowsInstaller.Installer");
```

Then, using the methods and properties, you will create additional objects and perform actions that are required as part of an installation program. Each application that exposes its functionality through one or more objects has an *object model*. This

object model is a pictorial means to show which objects are created from other objects. We will look at each of the object models for the three ActiveX components discussed in this chapter.

The Windows Installer Automation Interface

There are a number of objects that are exposed by the Windows Installer automation interface. Many of these objects can be used at build time and many can be used at run time. With InstallScript, the primary point of interest is the run-time use of Windows Installer objects. However you can use VBScript or JScript to manipulate the .msi file at build time. This manipulation would be performed in a post-build operation after InstallShield Developer had performed the build.

It is interesting to note that the InstallShield Developer project file is nothing more than an .msi file in disguise. Therefore you can use the Windows Installer automation interface to manipulate this project file. You could use this automation interface to perform operations on the project file prior to making the build. However, InstallShield Developer has its own automation interface for the project file and it provides functionality that the Windows Installer automation interface does not provide. Therefore you should actually use the InstallShield Developer automation interface for performing pre-build manipulation of the project file.

There are two areas of interest regarding the run-time use of the objects exposed by the Windows Installer engine. You can query the target system about the products and components that have already been installed or you can work with the database that is used for the installation that is in progress. This section discusses how to use Windows Installer objects to query the target system. Chapter 11 discusses how to work with the database that is active during an installation.

The Windows Installer Object Model

The hierarchy of the objects in the Windows Installer object model is shown in Figure 9-4.

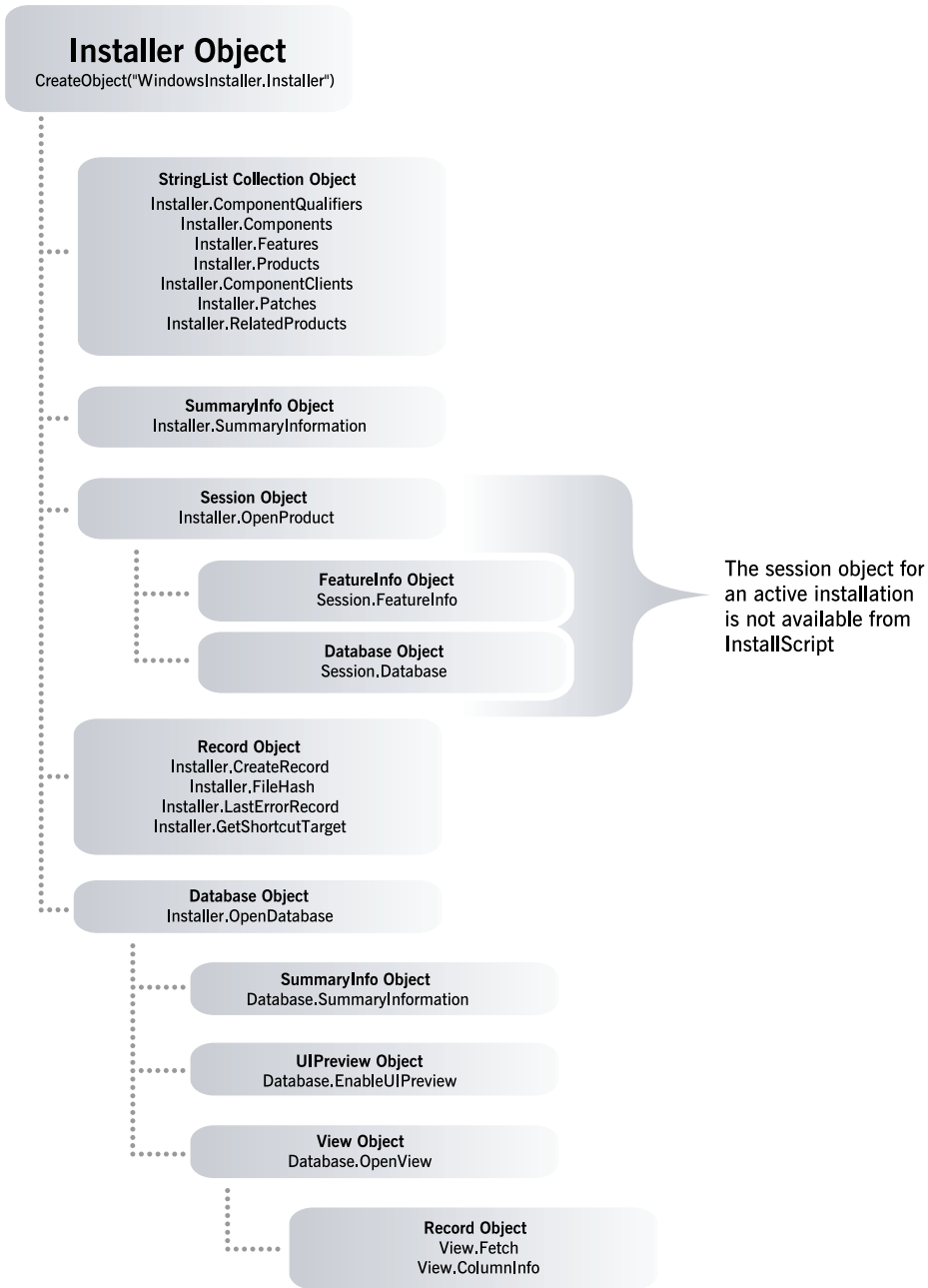


Figure 9-4: *The Windows Installer object model.*

As shown in Figure 9-4, the top-level object is the Installer object and all other objects are created from that object, with one exception. The Windows Installer engine, at the start of an installation, automatically creates the Session object. This particular object is only available to VBScript and JScript custom actions. You cannot access the Session object from InstallScript because it is not creatable. You can only implement COM objects in InstallScript that can be created using the function `CreateObject`.

Chapter 4 provides a detailed discussion of how the InstallScript engine and the Windows Installer engine run in different processes. Chapter 4 also goes into detail about how these two processes communicate with each other across the process boundary. When the Windows Installer is running InstallScript code it sees this code as being contained in a DLL.

It is outside the scope of this book to go into detail about all the methods and properties of all the objects shown in Figure 9-4. These properties and methods are documented in the Windows Installer help file that is available from the Help drop-down menu in InstallShield Developer.

However, this section does provide an overview of these objects and then works with several examples so you can understand the basics of using the Windows Installer automation interface. You will then work more with these objects in Chapter 11, which covers how to use InstallScript to create custom actions.

The Installer Object

The Installer object is the top-level object and you have to create this object before you can create any of the other objects. Create an Installer object by using the following statement in your script:

```
set objInstaller = CreateObject("WindowsInstaller.Installer");
```

There are 22 properties and 24 methods exposed by an Installer object. Of the 22 properties, 8 are used to create other objects and 14 are used to query the target system for information about installed products. Of the 24 methods, 7 are used to create other objects and 17 are used to perform actions relative to the target system. In general the properties and methods used to create other objects are covered

separately from the properties and methods that are used to interface with the target system.

Table 9-2 shows the properties and their descriptions that are used to query the target system for information. None of these properties are used to create other objects in the Windows Installer object hierarchy. This chapter covers the specific properties that are used to create other objects in the section that discusses that specific object.

Table 9-2: Installer Object Properties Used to Query Target System

Property	Description
ComponentPath	This is a read-only property that returns the key path to the specified component. If the key path is the name of a file in the component, the full path to the file is returned. If the key path is a registry entry, the registry entry is returned. If the key path is a folder, the full path to the folder is returned.
Environment	This is a read/write property that can be used to set the value of an environment variable or retrieve the value of an environment variable. If this property is used to set an environment variable, it does not persist beyond the end of the installation. To permanently set an environment variable, you need to use the Environment table.
FeatureParent	This property is read only and it returns the parent feature for a specified feature. If there is no parent feature, the value of this property is a NULL string.
FeatureState	This is a read-only property that returns the state of an installed feature. Feature install states can be Absent, Advertised, Run Locally, or Run From Source.

Table 9-2: Installer Object Properties Used to Query Target System (Continued)

Property	Description
FeatureUsageCount	This is a read-only property that returns the number of times that a product feature has been used. This value could be used to determine if a feature that is not used very much should be uninstalled during an upgrade.
FeatureUsageDate	This is a read-only property that returns the last date the specified product feature was used. The date is returned in the MS-DOS date format.
FileAttributes	This is a read-only property that returns the attributes of a file in a combined fashion. To get the individual attributes, you need to evaluate the number that is returned using a mask and some bitwise operations.
PatchInfo	This is a read-only property that returns the name of the locally cached package for a specified patch. This lets you know to what product a patch is applied whenever you perform a maintenance operation.
PatchTransforms	This is a read-only property that returns a semicolon delimited list of the transforms that are included in the specified locally cached patch package.
ProductInfo	This is a read-only property that returns a specified attribute for an installed product. Many of these attributes for a product are the values of properties that were set in the installation database.

Table 9-2: Installer Object Properties Used to Query Target System (Continued)

Property	Description
ProductState	This is a read-only property that returns the install state of a specified product.
QualifierDescription	This is a read-only property that returns the description of the specified qualified component. See Chapter 13 for a discussion of qualified components.
UILevel	This is a read/write property that specifies the type of user interface to be used when opening and processing subsequent packages within the current process space.
Version	This is a read-only property that returns the string representation of the current version of the Windows Installer. The string is returned in the following form: major.minor.build.update

In general you would access the values of the above properties from one of the event handlers in a Standard project or from a custom action in a Basic MSI project.

After looking at the available methods for making changes to or querying the target system, you will work through an example script that uses some of these properties and methods. Table 9-3 provides an overview of the Installer object methods that are used for other purposes than to create objects.

Table 9-3: Installer Methods Not Used to Create Objects

Method	Description
AddSource	This method is used to add an additional network source location for a product. This method supports <i>source resiliency</i> for an installed product. Source resiliency is a means used by the Windows Installer to identify a number of different locations where the source files for an application can be found.
ApplyPatch	This method applies a patch to each installed product to which the patch is applicable.
ClearSourceList	This method removes all of the network locations in the registry that indicates valid network sources for installed products. This method affects the source resiliency of an installed application.
CollectUserInfo	This method is run from an installed application and only the first time the application is used after installation. This method invokes a user interface that asks the user for basic information. The purpose of this method is to facilitate the implementation of an application-side security mechanism.
ConfigureFeature	This method is used to alter the installed state of a product feature. The possibilities are to have the feature advertised, installed locally, uninstalled, installed to run from source, or installed to its default location.
ConfigureProduct	This method is used to alter the installed state of a product. The possibilities are to have the product advertised, installed locally, uninstalled, installed to run from source, or installed to its default location.

Table 9-3: Installer Methods Not Used to Create Objects (Continued)

Method	Description
EnableLog	This method enables logging for any installations that are launched in the process that calls this method.
FileSize	This method returns the size of a specified file. This method uses the normal Win32 APIs to obtain the file size.
FileVersion	This method returns the version string or language string of a specified file. The format for versions is a string in "#.#.#.#" format. For a language, the value is the decimal language ID.
ForceSourceListResolution	This method forces the Windows Installer to search the source list for a valid product source the next time a source is needed. An example of this is when the Windows Installer performs an installation or a reinstallation, or when it needs the path for a component set to run from source.
InstallProduct	This method initiates the installation of a product. Depending on the command line used, this method can also be used to uninstall a product.
ProvideComponent	This method combines the functionality of the UseFeature, ConfigureFeature, and ComponentPath methods. It is used to perform any required installation of a component.
ProvideQualifiedComponent	This method returns the full path of the qualified component and performs any necessary installation.

Table 9-3: Installer Methods Not Used to Create Objects (Continued)

Method	Description
RegistryValue	This method reads information about a specified registry key or value.
ReinstallFeature	This method reinstalls features or corrects problems with installed features
ReinstallProduct	This method reinstalls a product or corrects installation problems in an installed product.
UseFeature	This method increments the usage count for a particular feature and returns the installation state for that feature.

The main purpose of most of the methods shown in Table 9-3 is to provide applications with self-repair capability. A few of them, however, can be used to implement functionality from an installation program. Some of these methods can be used during the application upgrade process. At this point, we will look at an example script that uses some of the properties and one or two of the methods described above. This example introduces the use of COM in InstallScript.

The first example demonstrates the use of several of the Installer object properties. The code for this example is shown in Figure 9-5. This example obtains information about the InstallShield Developer product from the information that is written to the registry.

The first thing to note about this program is that many of the properties exposed by the Installer object deal with products that are already installed. Because of this, the program needs to pass the value of the ProductCode property when accessing these properties. Also note that the program has to explicitly declare the values of the codes returned from some of the Installer object properties. Unfortunately the Windows Installer help does not document many of the values of these codes. These properties are designed for use in Visual Basic where these constants are available by referencing the Microsoft Windows Installer Object Library (a type library embedded as a

resource in msi.dll). The best way to obtain the value of these constants is to open a project in Visual Basic and reference this object library. After opening a project in Visual Basic, you can go to the Object Browser and find the value of all constants for the Windows Installer library.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates several
//                of the Installer object properties.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION      "Feedback"

// Product code for InstallShield Developer
#define PRODUCTCODE "{0F031DEC-3150-4503-9744-9431264BBA48}"
// Return constants used by the ProductState property
#define msiInstallStateBadConfig      -6
#define msiInstallStateInvalidArg    -2
#define msiInstallStateUnknown       -1
#define msiInstallStateAdvertised    1
#define msiInstallStateAbsent        2
#define msiInstallStateDefault       5

INT      iReturn;
STRING   szInstalledState, szProductName;
STRING   szInstallDate, szManufacturer;
OBJECT   objInstaller;

program

    try
        // Create an Installer object.
        set objInstaller =
            CreateObject("WindowsInstaller.Installer");

        // Obtain the install state of InstallShield Developer.
        iReturn = objInstaller.ProductState(PRODUCTCODE);

```

Figure 9-5: *Setup.rul that demonstrates several Installer object properties.*

```

// Display the install state of InstallShield Developer.
switch(iReturn)
  case msiInstallStateBadConfig:
    szInstalledState =
      "The configuration data is corrupt.";
  case msiInstallStateInvalidArg:
    szInstalledState =
      "An invalid parameter was passed to the function.";
  case msiInstallStateUnknown:
    szInstalledState =
      "The product is neither advertised nor installed.";
  case msiInstallStateAdvertised:
    szInstalledState =
      "The product is advertised but not installed.";
  case msiInstallStateAbsent:
    szInstalledState =
      "The product is installed for a different user.";
  case msiInstallStateDefault:
    szInstalledState =
      "The product is installed for the current user.";
  default:
    szInstalledState = "Unknown installed state.";
endswitch;

szProductName = objInstaller.ProductInfo
  (PRODUCTCODE, "InstalledProductName");
szInstallDate = objInstaller.ProductInfo
  (PRODUCTCODE, "InstallDate");
szManufacturer = objInstaller.ProductInfo
  (PRODUCTCODE, "Publisher");

// Display the product information.
sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
  "Product code: %s\nProduct name: %s\n" +
  "Manufacturer: %s\nInstalled state: %s\n" +
  "Install date: %s",
  PRODUCTCODE, szProductName, szManufacturer,
  szInstalledState, szInstallDate);
catch
  sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
    "Exception thrown.");
endcatch;

set objInstaller = NOTHING;

endprogram

```

Figure 9-5: *Continued.*

The program shown in Figure 9-5 uses two Installer object properties, `ProductState` and `ProductInfo`. It accesses these properties by first creating an Installer object and then calling these properties to return the desired values. Note that every time that you create an object you must use the `set` keyword. This program uses the properties exposed by the Windows Installer to query the system for information about products installed using the Windows Installer. In general, using the automation interface properties and methods from `InstallScript` is limited to gathering information about the system.

The Windows Installer object model in Figure 9-4 shows the five objects that can be created from the Installer object. `InstallScript` can only really effectively use the `StringList` and the `Record` objects. The `SummaryInfo` and `Database` objects are used for authoring a database. The `Session` object is not available to an `InstallScript` program because it is created by and available only to the Windows Installer process that runs an installation. The `Session` object, as already mentioned, is only accessible to `VBScript` and `JScript` custom actions.

The following sections take a closer look at the `StringList` and `Record` objects since they are the only objects, other than the Installer object, that have any use to an `InstallScript` program or custom action. You will then work through another `InstallScript` example that uses these objects.

The StringList Object

A `StringList` object defines a collection of strings that identify various entities. There are seven different `StringList` objects that can be created using an Installer object property. These properties are discussed in Table 9-4.

Table 9-4: Installer Properties Used to Create StringList Objects

Property	Description
<code>Installer.ComponentClients</code>	This read-only property returns a <code>StringList</code> object that enumerates the set of clients for the specified component. The client of a component is a product and this product is defined by the value of its <code>ProductCode</code> property.

Table 9-4: Installer Properties Used to Create StringList Objects (Continued)

Property	Description
Installer.ComponentQualifiers	This read-only property returns a <code>StringList</code> object enumerating the set of registered qualifiers for a specified component. This has to do with qualified components or an array of components.
Installer.Components	This read-only property returns a <code>StringList</code> object that enumerates all the installed components for all installed products.
Installer.Features	This read-only property returns a <code>StringList</code> object that enumerates all the features for a specified product. This property returns the list of features in any order and not necessarily the order in which the features were installed.
Installer.Patches	This read-only property returns a <code>StringList</code> object that enumerates all the patches applied to a specified product.
Installer.Products	This read-only property returns a <code>StringList</code> object that enumerates all the products that have been installed for the current user or for the machine. This property returns the list of products in any order and not necessarily the order in which the products were installed.
Installer.RelatedProducts	This read-only property returns a <code>StringList</code> object that enumerates the set of all installed products associated with a specified <code>UpgradeCode</code> .

After you create one of the `StringList` objects (Table 9-4), you have to do something with it. The primary use for a `StringList` object is to iterate through it looking for any needed information. Two properties are available to enable traversing a `StringList` object (Table 9-5). These properties allow you to find out how many strings are in a `StringList` object and an index that allows you to access each of these strings.

Table 9-5: The `StringList` Object Properties

Property	Description
Count	This read-only property returns the number of items in a <code>StringList</code> Object. The usage of this property is as follows: <code>iItems = objStringList.Count;</code>
Item	This read-only property returns a value in the <code>StringList</code> object. The usage of this property is normally inside a loop and is as follows: <code>szString = objStringList.Item(i);</code> Looping proceeds <code>iItems</code> number of times in order to access all values that are enumerated by the <code>StringList</code> object. The index <code>i</code> for <code>Item</code> is 0 based.

The concept of a `StringList` object is quite simple. Accessing the contents of a `StringList` object is similar to accessing the contents of an array using a numerical index. There are no methods that are available to work with a `StringList` object.

The Record Object

For the `Record` object, there are three `Installer` object methods and one property that will create a record. One of these methods creates an empty record, but the other two methods and one property return values in the fields of records of defined sizes. The `Installer` object methods and properties that create record objects are described in Table 9-6.

The Record object is a container object for holding and transferring a variable number of values. Fields within the record are numerically indexed and can contain strings, integers, objects, and Null values. Fields beyond the allocated record size are treated as having permanently Null values. Field number 0 is reserved so the actual values of interest are contained in fields that start with and index of 1.

Table 9-6: Methods or Properties for Creating Record Objects

Method/Property	Description
Installer.CreateRecord (Method)	This method returns a new Record object that has the specified number of fields. With Record objects, the indexing of fields is 1-based because there is a 0-index field that is used for special purposes and this field is not part of the field count.
Installer.FileHash (Method)	This method takes the path to a file and returns a 128-bit hash of that file. The file hash information is returned as a Record object. The entire 128-bit file hash is returned as four 32-bit integer data property fields. A file hash is used by the Windows Installer to determine whether a non-versioned file should be overwritten during an installation.
Installer.GetShortcutTarget (Property)	This property examines a specified shortcut and returns in a Record object with three fields containing the product code, the feature name, and the component code.
Installer.LastErrorRecord (Method)	This method returns a Record object that contains the error parameters for the most recent error generated by particular methods.

To work with the records created using the above methods, there are both properties and methods that are exposed by the Record object. There are five properties and four methods. The five properties of a Record object are discussed in Table 9-7 and

the four methods are discussed in Table 9-8. Remember that the main purpose of the Record object is to hold and/or transfer data.

Table 9-7: The Record Object Properties

Property	Description
DataSize	This read-only property returns the size of the data for the designated field. If the data is a stream, the stream length in bytes is returned. If the data is a string, the string length without the terminating Null is returned. If the data is an integer, the value 4 is returned. If the data is Null, 0 is returned.
FieldCount	This read-only property returns the number of fields in the record. Trying to read fields beyond this count returns a Null value. Trying to write to fields beyond this count fails.
IntegerData	This read/write property transfers 32-bit integer data into or out of a specified field within the record.
IsNull	This read-only property returns True if the indicated field is Null, and False if the field contains data.
StringData	This read/write property transfers string data into or out of a specified field within the record. If an integer or object has been stored, its string value is returned.

In Table 9-7 you can see that all five of the Record properties can be used to retrieve values from a Record object but only two of these properties can set the value of a field in a record. Only integer or string data can be set using a property. Setting the other types of data that a record can contain requires the use of methods.

Table 9-8: The Record Object Methods

Method	Description
ClearData	This clears the data in all fields by setting them to Null. Any objects stored in the fields are released.
FormatText	This method formats fields according to the template in field 0.
ReadStream	This method reads a specified number of bytes from a record field holding stream data.
SetStream	This method copies the content of the specified file into the designated record field as stream data.

A Script Example

This section provides a final example of using the Automation Interface to the Windows Installer. This example works primarily with the `StringList` object and the `Record` object.

As with the previous example, this example works with various aspects of the InstallShield Developer installation. The code for this example is shown in Figure 9-6. In this example, the program first examines the number and names of the features that compose InstallShield Developer. Secondly, the program creates a file hash for the `Funcwiz.ini` file. File hashing is a means to determine if non-versioned files are different.

To get the features for InstallShield Developer, you use the `Features` property of the `Installer` object. This gives a `StringList` object that contains the names of all the features for the `ProductCode` that was passed as its argument. The program then gets the number of features in the `StringList` object, using the `Count` property, so that it can loop through this list and create a display string to use in the call to the `SprintfBox` function. To loop through the `StringList` of feature names, the

program uses the Index property of this object, passing this property the index of the string that it wants to retrieve.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates the
//                StringList and Record objects.
//
////////////////////////////////////
#include "ifx.h"

#define CAPTION      "Feedback"

// Product code for InstallShield Developer
#define PRODUCTCODE "{F91CEC68-2512-410B-93DC-4AA79F1EF7B7}"

INT      i, iCnt;
LONG     dwData(4);
STRING   szFeature, szDisplay;
OBJECT   objInstaller, objRecord, objStringList;

program
  try
    // Create an Installer object.
    set objInstaller =
      CreateObject("WindowsInstaller.Installer");

    // Create a StringList object that contains the names of
    // all the features for InstallShield Developer
    set objStringList = objInstaller.Features(PRODUCTCODE);

    // Get the number of features.
    iCnt = objStringList.Count;

    // Loop through all the features and add them to
    // a display variable.
    for i=0 to iCnt-1
      szFeature = objStringList.Item(i);
      if(i != iCnt-1) then
        szDisplay = szDisplay + szFeature + ", ";
      else
        szDisplay = szDisplay + szFeature;
      endif;
    endfor;
  
```

Figure 9-6: *Setup.rul* file that demonstrates the *StringList* and *Record* objects.

```

// Display the number and names of the features
// in the InstallShield Developer product.
SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
    "InstallShield Developer Features\n\n" +
    "Number of features = %d\n%s",
    iCnt, szDisplay);

// Create a record object that holds a file hash
// for the Funcwiz.ini file.
set objRecord = objInstaller.FileHash("C:\\Program Files" +
    "\\InstallShield\\Developer\\Support\\Funcwiz.ini", 0);

// Extract the value in each field of the
// file hash record.
dwData(0) = objRecord.IntegerData(1);
dwData(1) = objRecord.IntegerData(2);
dwData(2) = objRecord.IntegerData(3);
dwData(3) = objRecord.IntegerData(4);

// Display the values of the file hash.
SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
    "Field 1: %lu\nField 2: %lu\nField 3: %lu\nField 4: %lu",
    dwData(0), dwData(1), dwData(2), dwData(3));

catch
    SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
        "Exception thrown.");
endcatch;

set objRecord = NOTHING;
set objStringList = NOTHING;
set objInstaller = NOTHING;

endprogram

```

Figure 9-6: *Continued.*

The value of the ProductCode property for InstallShield Developer used in the above example might be different than what is in the version of InstallShield Developer you are using. Before running this example you should open up the .msi file for InstallShield Developer using Orca and get the value of the ProductCode property from the Property table

To create the Record object that contains the hash value for the Funcwiz.ini file, the program uses the FileHash method of the Installer object. Once you have this object, you can access each of the fields in the record using the IntegerData property of the Record object. The program sets the four elements of an array to the value of the

four fields in the record and then displays these values as unsigned long values. As in the previous program, you can then set all the objects that were created to NOTHING, which is the same as setting them to NULL.

The Scripting Run-Time Objects

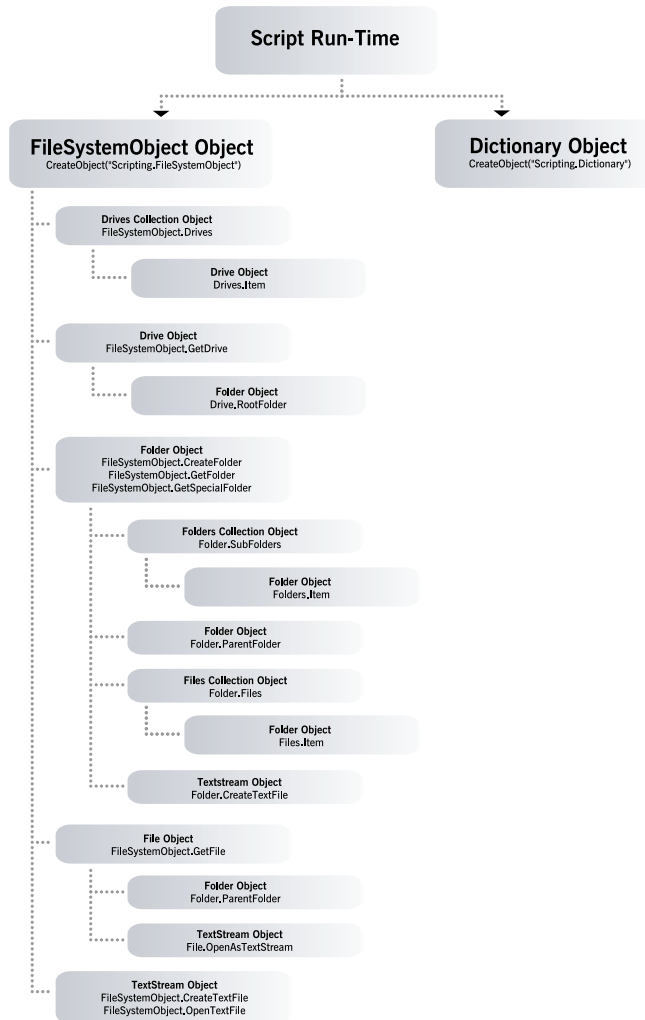


Figure 9-7: *The Scripting Run-Time object model.*

The objects available from the Scripting Run-Time Library provide many methods and properties that provide access to the Windows file-system. The Scripting Run-Time Library is available in the file `scrrun.dll` (a system file). The main object covered in this chapter is the `FileSystemObject` object, but we will also discuss the `Dictionary` object. The `Dictionary` object is a way to create an associative array in `InstallScript`. The object model for the scripting run-time functionality found in the `scrrun.dll` file is shown in Figure 9-7.

As shown in Figure 9-7, the `FileSystemObject` object provides a number of objects that deal with drives, folders, and files. The `Dictionary` object is just an object by itself with no hierarchy of objects to be created. This chapter provides a brief overview of these objects, so you should consult the full documentation from the Microsoft Web site for additional information. To get the full documentation of version 5.6 of the Windows Script Host, do the following:

1. Access the MSDN Web site.
2. Click on the Downloads link.

Search for the string "Windows Script Host". This documentation is appropriate for the material in this section and in the next section.

The `FileSystemObject` Object

The `FileSystemObject` object provides a means to access the file system of the target system. Unlike with the Windows Installer automation interface, all the objects, methods, and properties are available from `InstallScript`. To create a `FileSystemObject` object, use a line of code such as follows:

```
set fso = CreateObject("Scripting.FileSystemObject");
```

In the `FileSystemObject` object, there are four objects and three collections. This discussion refers to collections as objects because they have properties and methods. Table 9-9 discusses the creation of each of the nine objects that are available from the `FileSystemObject` object. The discussion in Table 9-9 assumes that you have already created a `FileSystemObject` object with the name `fso`.

Table 9-9: The Creation of the FileSystemObject Objects

Object	Description
Drives (Collection)	<p>A Drives collection is created using the Drives property of the FileSystemObject object. This is implemented in InstallScript in the following manner.</p> <pre data-bbox="501 472 786 495">set dc = fso.Drives;</pre> <p>This collection object does not have a numerical index that can be used to traverse it. In InstallScript this presents a special problem since the language does not have a For Each...Next construct. We will look at an example of how to handle this problem for a Drives collection later in this chapter.</p>
Drive	<p>A Drive object identifies a drive in a specified path. This object is created using the GetDrive method of the FileSystemObject object. A Drive object is created in InstallScript using a statement similar to the following:</p> <pre data-bbox="501 887 958 910">set d = fso.GetDrive(drivespec);</pre> <p>The <i>drivespec</i> argument can be a drive letter "C", a drive letter with a colon appended "C:", a drive letter with a colon and path separator "C:\", or a network share specification "\\computer\share".</p> <p>A Drive object can also be created using the Item property of the Drives collection.</p>
Files (Collection)	<p>A Files collection is created using the Files property of the Folder object. This is implemented in InstallScript in the following manner.</p> <pre data-bbox="501 1330 1058 1377">set folder = fso.GetFolder(folderspec); set files = folder.Files;</pre> <p>In InstallScript it is not possible to traverse a Files collection. The only thing that you can do is check to see if a certain file exists in the collection.</p>

Table 9-9: The Creation of the FileSystemObject Objects (Continued)

Object	Description
File	<p>A File object identifies a file for which you want to get the attributes. This object is created using the GetFile method of the FileSystemObject object. A File object is created in InstallScript using a statement similar to the following:</p> <pre data-bbox="501 508 972 530">set file = fso.GetFile(filespec);</pre> <p>The <i>filespec</i> argument can be an absolute or a relative path to a file.</p> <p>A File object can also be created using the Item property of the Files collection.</p>
Folders (Collection)	<p>A Folders collection is created using the SubFolders property of the Folder object. This is implemented in InstallScript in the following manner:</p> <pre data-bbox="501 878 1058 931">set folder = fso.GetFolder(folderspec); set folders = folder.SubFolders;</pre> <p>In InstallScript it is not possible to traverse a Folders collection. The only thing that you can do is to check to see if a certain folder exists in the collection.</p>
Folder	<p>A Folder object identifies a folder for which you want to get the attributes. This object can be created using the GetFolder method of the FileSystemObject object. A Folder object using this method is created in InstallScript using a statement similar to the following:</p> <pre data-bbox="501 1284 1058 1307">set folder = fso.GetFolder(folderspec);</pre> <p>The <i>folderspec</i> argument can be an absolute or a relative path to a folder.</p> <p>A Folder object can be created by other methods and properties, as shown in Figure 9-7.</p>

Table 9-9: The Creation of the FileSystemObject Objects (Continued)

Object	Description
TextStream	<p>A TextStream object identifies a text file to which you want sequential access. This object can be created using the CreateTextFile method of the FileSystemObject object or the Folder object. A TextStream object using this method is created in InstallScript using a statement similar to the following:</p> <pre data-bbox="501 578 1219 627">set textfile = fso.CreateTextFile(filename, overwrite, unicode);</pre> <p>The <i>filename</i> argument identifies the name to create. It can specify a path. If a path is not specified, the file is created in the current directory. The <i>overwrite</i> argument is optional and it specifies whether an existing file can be overwritten. To permit a file to be overwritten, set this argument to TRUE. The default is FALSE. The <i>unicode</i> argument defines whether an ASCII or a Unicode text file is to be created. To have a Unicode file created, set this argument to TRUE. The default is FALSE.</p> <p>Additionally a TextStream object can be created using the OpenTextFile method of the FileSystemObject or the OpenAsTextStream method of the File object.</p>

So far we have seen how to create a FileSystemObject and then how to use this object to create all the other objects in the object model hierarchy. The next section looks at the other methods that are exposed by the FileSystemObject object directly.

FileSystemObject Object Methods

The FileSystemObject object exposes only one property and that is the property already discussed in Table 9-9 for creating a Drives collection. Table 9-10 shows the methods of the FileSystemObject object that are not used to create other objects.

Table 9-10: The Methods of the FileSystemObject Object

Method	Description
BuildPath	<p>This method appends a name to a path. Building path names for files and folders is a very common operation in any installation program. A call to this method in InstallScript might look like this:</p> <pre>newpath = fso.BuildPath(path, name);</pre> <p>The <i>path</i> argument is either an absolute or relative path that either exists or does not exist. The <i>name</i> argument is the name of a folder or a file. A path separator is inserted if necessary. This performs much the same functionality that the path concatenate operator (^) does in InstallScript. The path buffer functions in InstallScript also server the purpose of building paths.</p>
CopyFile	<p>This method copies one or more files from one location to another. A call to this method in InstallScript might look like this:</p> <pre>fso.CopyFile(source, destination, [overwrite]);</pre> <p>The <i>source</i> argument specifies the file or files to be copied. This string can include wildcard characters. The <i>destination</i> argument specifies the location where the files are to be copied. If this destination is the name of a file, it will be overwritten depending on the <i>overwrite</i> argument. If the destination is a folder, it must terminate with a path separator. The <i>overwrite</i> argument is optional and, by default, is set to TRUE. The functionality of this method is somewhere between the CopyFile and the XCopyFile functions found in InstallScript.</p>

Table 9-10: The Methods of the FileSystemObject Object (Continued)

Method	Description
CopyFolder	<p>This method copies one or more folders and their contents from one location to another. A call to this method in InstallScript might look like this:</p> <pre>fso.CopyFolder(source, destination, [overwrite]);</pre> <p>The <i>source</i> argument specifies the folder or folders to be copied. This string can include wildcard characters. The <i>destination</i> argument specifies the location where the folders are to be copied. If the destination is a folder terminated with a path separator, it is assumed that this folder exists. The <i>overwrite</i> argument is optional and, by default, is set to TRUE. This argument determines whether folders are to be overwritten if they already exist. In InstallScript you can use the XCopyFile function to perform the same operation.</p>
DeleteFile	<p>This method deletes one or more files. A call to this method in InstallScript might look like this:</p> <pre>fso.DeleteFile(filespec, [force]);</pre> <p>The <i>filespec</i> argument names the file to be deleted. This argument can include wildcard characters in the last component of the path. The optional <i>force</i> argument allows for the deletion of a read-only file if this argument is set to TRUE. The default for this argument is FALSE. You can delete files using the DeleteFile function in InstallScript. However, this method is more powerful than the DeleteFile function in InstallScript because it can remove read-only files.</p>

Table 9-10: The Methods of the FileSystemObject Object (Continued)

Method	Description
DeleteFolder	<p>This method deletes one or more folders and their contents. A call to this method in InstallScript might look like this:</p> <pre data-bbox="601 469 1139 495">fso.DeleteFolder(folderspec, [force]);</pre> <p>The <i>folderspec</i> argument names the folders to be deleted. This argument can include wildcard characters in the last component of the path. The optional <i>force</i> argument allows for the deletion of a read-only file if this argument is set to TRUE. The default for this argument is FALSE. This method is similar to the DeleteDir function in InstallScript. There are differences in functionality, so which should be used depends on the desired action. For example the DeleteDir function cannot delete a read-only folder but the DeleteFolder method can. Also, you can use wild cards in the <i>folderspec</i> argument in the call to the DeleteFolder method but you need to specify an absolute path to the folder to be deleted in the DeleteDir function.</p>
DriveExists	<p>This method returns TRUE if the specified drive exists and FALSE if it does not exist. A call to this method in InstallScript might look like this:</p> <pre data-bbox="601 1210 1125 1236">bReturn = fso.DriveExists(drivespec);</pre> <p>The <i>drivespec</i> argument specifies the drive letter or the full UNC path to a shared drive. The return value of TRUE for a drive with removable media does not indicate that there is media in the drive. You need to use the IsReady property of the Drive object to determine this. The DriveExists method is similar to InstallScript's ExistsDisk function.</p>

Table 9-10: The Methods of the FileSystemObject Object (Continued)

Method	Description
FileExists	<p>This method returns TRUE if the specified file exists and FALSE if it does not exist. A call to this method in InstallScript might look like:</p> <pre>bReturn = fso.FileExists(filespec);</pre> <p>The <i>filespec</i> argument specifies the absolute or relative path to the file whose existence is being determined. If only the file name is specified, only the current directory is searched. In InstallScript you can perform the same operation using the IS function. The FileExists method can take both a relative path and an absolute path to the file being checked. The IS function can only take an absolute path to the file.</p>
FolderExists	<p>This method returns TRUE if the specified folder exists and FALSE if it does not exist. A call to this method in InstallScript might look like:</p> <pre>bReturn = fso.FolderExists(folderspec);</pre> <p>The <i>folderspec</i> argument specifies the absolute or relative path to the folder whose existence is being determined. If only the folder name is specified, only the current directory is searched. This method is similar to InstallScript's ExistsDir function.</p>
GetAbsolutePathName	<p>This method returns the complete and unambiguous path specification for the input location. A call to this method in InstallScript might look like:</p> <pre>szPath = fso.GetAbsolutePathName(pathspec);</pre> <p>The <i>pathspec</i> argument can be any partial path specification that is related to the current directory. The return value is the absolute path to this location. There is no similar function in InstallScript.</p>

Table 9-10: The Methods of the FileSystemObject Object (Continued)

Method	Description
GetBaseName	<p>This method returns the last element in a path minus any file extension if the last element is a file. A call to this method in InstallScript might look like:</p> <pre>szBase = fso.GetBaseName(pathspec);</pre> <p>The <i>pathspec</i> argument defines the string that identifies a folder or file location. This location does not have to exist because this method only treats this as a string in the form of a path. There is no similar function in InstallScript.</p>
GetDriveName	<p>This method returns the drive letter element in a path specification string. A call to this method in InstallScript might look like:</p> <pre>szDrive = fso.GetDriveName(pathspec);</pre> <p>The <i>pathspec</i> argument defines the string that identifies a folder or file location. This location does not have to exist because this method only treats this as a string in the form of a path. This method is similar to InstallScript's GetDisk function. Both this method and the GetDisk function work with standard paths as well as with UNC path names.</p>
GetExtensionName	<p>This method returns the extension name of the last element in a path specification string. A call to this method in InstallScript might look like:</p> <pre>szExt = fso.GetExtensionName(pathspec);</pre> <p>The <i>pathspec</i> argument defines the string that identifies a folder or file location. This location does not have to exist because this method only treats this as a string in the form of a path. There is no similar function in InstallScript.</p>

Table 9-10: The Methods of the FileSystemObject Object (Continued)

Method	Description
GetFileName	<p>This method returns the last element in a path including any file extension if the last element is a file. A call to this method in InstallScript might look like:</p> <pre>szFile = fso.GetFileName(pathspec);</pre> <p>The <i>pathspec</i> argument defines the string that identifies a folder or file location. This location does not have to exist because this method only treats this as a string in the form of a path. There is no similar function in InstallScript.</p>
GetFileVersion	<p>This method returns the version number of the specified file. A call to this method in InstallScript might look like:</p> <pre>szVersion = fso.GetFileVersion(pathspec);</pre> <p>The <i>pathspec</i> argument defines the location of the file for which you want to retrieve the version. This method is similar to InstallScript's VerGetFileVersion function. The main difference is that this method can accept both an absolute path as well as a relative path to the file whose version is being retrieved.</p>
GetParentFolderName	<p>This method returns the parent of the last component in a specified path. A call to this method in InstallScript might look like:</p> <pre>szParent = fso.GetParentFolderName(pathspec);</pre> <p>The <i>pathspec</i> argument defines the string that identifies a folder or file location. This location does not have to exist because this method only treats this as a string in the form of a path. There is no similar function in InstallScript.</p>

Table 9-10: The Methods of the FileSystemObject Object (Continued)

Method	Description
GetTempName	<p>This method returns a randomly generated temporary file or folder name that can be used for performing operations that require a temporary file or folder. A call to this method in InstallScript might look like:</p> <pre>szTemp = fso.GetTempName();</pre> <p>There is no similar function in InstallScript.</p>
MoveFile	<p>This method moves one or more files from one location to another. A call to this method in InstallScript might look like:</p> <pre>fso.MoveFile(source, destination);</pre> <p>The <i>source</i> argument is the absolute or relative path to the file or files that are to be moved. The <i>destination</i> argument is the absolute or relative specification of the location to where the files are to be moved. There is no comparable function in InstallScript.</p>
MoveFolder	<p>This method moves one or more folders from one location to another. A call to this method in InstallScript might look like:</p> <pre>fso.MoveFolder(source, destination);</pre> <p>The <i>source</i> argument is the absolute or relative path to the folder or folders that are to be moved. The <i>destination</i> argument is the absolute or relative specification of the location to where the folders are to be moved. There is no comparable function in InstallScript.</p>

The example shown in Figure 9-8 uses some of the methods described in Table 9-10 to manipulate and gather information from the target system.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates several of the
//                methods exposed by the FileSystemObject object
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION      "Feedback"
#define PATHSPEC     "C:\\WINNT\\system32\\dllcache"

STRING  szNewPath, szFileName, szParentFolder;
STRING  szAbsolutePath, szFileVersion, szTemp;
BOOL    bExists;
OBJECT  fso;

program

    // Set the current directory to the location
    // specified by the PATHSPEC constant.
    ChangeDirectory(PATHSPEC);

    try
        // Create a FileSystemObject object.
        set fso = CreateObject("Scripting.FileSystemObject");

        // Add a file name to the PATHSPEC location.
        szNewPath = fso.BuildPath(PATHSPEC, "Shell32.dll");

        // Check to see if the new path exists.
        bExists = fso.FileExists(szNewPath);

        // If the file exists, get information
        // about this file and its location.
        if(bExists) then

            szFileName = fso.GetFileName(szNewPath);

            // This is why we needed to set the current directory.
            szAbsolutePath = fso.GetAbsolutePathName(szFileName);
            szParentFolder = fso.GetParentFolderName(szAbsolutePath);
            szFileVersion = fso.GetFileVersion(szAbsolutePath);
            szTemp = fso.GetTempName();

```

Figure 9-8: *Setup.rul* that shows several of the methods exposed by the *FileSystemObject* object.

```

        // Display the information that was retrieved.
        sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
            "File name: %s\nAbsolute path: %s\n" +
            "Parent Folder: %s\nFile version: %s\n" +
            "Temp folder: %s", szFileName,
            szAbsolutePath, szParentFolder,
            szFileVersion, szTemp);
    else
        sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
            "Path does not exist.");
    endif;

catch
    sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
        "Exception thrown.");
endcatch;

endprogram

```

Figure 9-8: *Continued.*

First, the program uses the `CurrentDirectory InstallScript` function to set the current directory to the value defined by the `PATHSPEC` constant. This is necessary because the example uses a method that relates to finding the absolute path of an element that is in the current directory. After creating the `FileSystemObject` object, the program adds a file name to the path defined by the `PATHSPEC` constant. It then verifies that the file exists. If it exists, the program executes several of the methods that allow you to retrieve information from the file system. In this chapter, none of the program examples make changes to the target system. This is discussed when we cover custom actions in Chapter 11.

Drive Object Properties

The Drive object does not expose any methods but it does provide a number of valuable properties (Table 9-11). A Drive object can be created in InstallScript using the following statements:

```

set fso = CreateObject("Scripting.FileSystemObject");
szDriveName = fso.GetDriveName(PATHSPEC);
set do = fso.GetDrive(szDriveName);

```

Table 9-11: Drive Object Properties

Property	Description
AvailableSpace	<p>This read-only property returns the amount of space available to a user on the specified drive or network share. The available space returned by this property may be less than the total free space if the computer operating system supports quotas. The available space is returned as a string, giving the space in bytes. To get the available space in megabytes, use an InstallScript statement similar to the following:</p> <pre data-bbox="596 654 1029 702">Value = do.AvailableSpace; iAvailSpace = Value/(1024*1024);</pre> <p>In these statements the variable <code>Value</code> is declared as type <code>VARIANT</code>. This works only if the available space is less than 4 GB.</p>
DriveLetter	<p>This read-only property returns the drive letter of a physical local drive or a network share. To get the drive letter, use an InstallScript statement similar to the following:</p> <pre data-bbox="596 1024 1015 1046">szDriveLetter = do.DriveLetter;</pre> <p>The drive letter returned by this property does not include the colon (:). If Drive object identifies a network share that does not map to a drive letter then the return will be a <code>NULL</code> string.</p>
DriveType	<p>This read-only property returns a value indicating the type of a specified drive. To get the drive type, use an InstallScript statement similar to the following statement:</p> <pre data-bbox="596 1404 879 1425">iType = do.DriveType;</pre> <p>When you get the number that indicates the drive type, you can set a string variable to a proper display value for the drive type.</p>

Table 9-11: Drive Object Properties (Continued)

Property	Description
FileSystem	<p>This read-only property returns the type of file system in use for the specified drive. To get the file system, use an InstallScript statement similar to the following statement:</p> <pre data-bbox="596 508 986 530">szFileSystem = do.FileSystem;</pre> <p>The possible values returned by this property are FAT, NTFS, and CDFS.</p>
FreeSpace	<p>This read-only property returns the amount of free space available to a user on the specified drive or network share. The available space that is returned by this property may be more than the total available space if the computer operating system supports quotas. The free space is returned as a string, giving the space in bytes. To get the free space in megabytes, use an InstallScript statement similar to the following:</p> <pre data-bbox="596 954 1015 1003">Value = do.FreeSpace; iFreeSpace = Value/(1024*1024);</pre> <p>In these statements the variable <code>Value</code> is declared as type <code>VARIANT</code>. This works only if the free space is less than 4 GB.</p>
IsReady	<p>This read-only property returns <code>TRUE</code> if the specified drive is ready and <code>FALSE</code> if the drive is not ready. To see if a drive is ready, use an InstallScript statement similar to the following:</p> <pre data-bbox="596 1324 865 1347">bReady = do.IsReady;</pre> <p>When the drive is for removable media, the value of <code>bReady</code> is <code>TRUE</code> only if the drive contains media.</p>

Table 9-11: Drive Object Properties (Continued)

Property	Description
Path	<p>This read-only property returns the path for a specified drive. To get the drive path, use an InstallScript statement similar to the following:</p> <pre data-bbox="596 472 825 493">szPath = do.Path;</pre> <p>Except for network share situations, this is not very useful. It returns the drive letter with the colon (:).</p>
SerialNumber	<p>This read-only property returns the serial number used to uniquely identify a disk volume. To get the volume serial number, you would use an InstallScript statement similar to the following:</p> <pre data-bbox="596 777 1029 799">iSerialNumber = do.SerialNumber;</pre> <p>The value returned is the same value that appears in the Command Prompt when you execute the Dir command.</p>
ShareName	<p>This read-only property returns the network share name for a specified drive. To get the network share name, use an InstallScript statement similar to the following:</p> <pre data-bbox="596 1121 961 1143">szShareName = do.ShareName;</pre> <p>If the Drive object does not refer to a network drive, this property returns a NULL string.</p>
TotalSize	<p>This read-only property returns the total space in bytes of a drive or network share. This is not very useful in InstallScript since it cannot represent a number greater than 4 GB.</p>

Table 9-11: Drive Object Properties (Continued)

Property	Description
VolumeName	<p>This read/write property returns or sets the volume name of a drive. To get the volume name, use an InstallScript statement similar to the following:</p> <pre>szVolumeName = do.VolumeName;</pre> <p>Trying to set a volume name without having sufficient privileges generates an exception.</p>

This next program uses some of the Drive object properties (Figure 9-9). The key to this program is to first create a FileSystemObject object and then use this object to obtain the drive name from a path using the GetDriveName method. Then, the program passes this drive name to the GetDrive method of the FileSystemObject object and creates a Drive object. From there, the program uses the various Drive object properties to retrieve information about the drive.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates several of the
//                properties exposed by the Drive object
//
////////////////////////////////////
#include "ifx.h"

#define CAPTION      "Feedback"
#define PATHSPEC     "C:\\WINNT\\system32\\dllcache"

INT      iType, iSerialNumber;
BOOL     bReady;
STRING   szDriveName, szDriveType;
STRING   szFileSystem, szReady;
VARIANT Value;
OBJECT   fso, do;

```

Figure 9-9: *Setup.rul showing the use of some Drive object properties.*

```

program

    try
        // Create a FileSystemObject object.
        set fso = CreateObject("Scripting.FileSystemObject");
        szDriveName = fso.GetDriveName(PATHSPEC);

        // Create a Drive object.
        set do = fso.GetDrive(szDriveName);

        // Get the drive type and convert
        // it to a display string.
        iType = do.DriveType;
        switch(iType)
            case 0:
                szDriveType = "Unknown type";
            case 1:
                szDriveType = "Removable drive";
            case 2:
                szDriveType = "Fixed drive";
            case 3:
                szDriveType = "Network drive";
            case 4:
                szDriveType = "CD-ROM drive";
            case 5:
                szDriveType = "RAM Disk";
        endswitch;

        // Get the type of file system.
        szFileSystem = do.FileSystem;

        // Check to see if the drive is ready
        // and then retrieve the serial number.
        if(do.IsReady) then
            iSerialNumber = do.SerialNumber;
        endif;

        // Check to see if the drive is ready
        // and then create a display string.
        if(do.IsReady) then
            szReady = "The drive is ready for use.";
        else
            szReady = "The drive is not ready for use.";
        endif;
    
```

Figure 9-9: *Continued.*


```

        // Display the information about the drive.
        sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
            "Drive Name: %s\nDrive Type: %s\nFile System: %s\n" +
            "Serial Number: %lu\nStatus: %s",
            szDriveName, szDriveType, szFileSystem,
            iSerialNumber, szReady);

    catch
        sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
            "Exception thrown.");
    endcatch;

    // Set the objects to NULL
    set fso = NOTHING;
    set do = NOTHING;

endprogram

```

Figure 9-9: *Continued.*

Folder and File Object Properties and Methods

The Folder and File objects expose the same properties and methods. These properties and methods allow you to work with individual folders and files. The 12 properties for the Folder object and the 11 properties for the File object are described in Table 9-12. The properties and methods discussed here are those that do not create other objects.

A Folder object and File object can be created in InstallScript as follows:

```

set fso = CreateObject("Scripting.FileSystemObject");
szFolderName = "C:\\NewFolder";
set fldr = fso.CreateFolder(szFolderName);
szFileSpec = "C:\\WINNT\\system32\\dllcache\\Shell132.dll"
set f = fso.GetFile(szFileSpec);

```

Using the `GetFolder` or the `GetSpecialFolder` methods of the `FileSystemObject` object can also create a Folder object.

Table 9-12: Folder and File Object Properties

Property	Description
Attributes	<p>This property sets or returns the attributes of files or folders. Depending on the attribute, this property is either read/write or read-only. For a list of the attributes, see the example program in Figure 9-10. To get the attribute of a file or folder, you would use an InstallScript statement similar to the following type:</p> <pre data-bbox="601 578 951 601">iAttrib = fldr.Attributes;</pre> <p>To set an attribute, for example to make a folder hidden, use an InstallScript statement as follows:</p> <pre data-bbox="601 710 868 733">fldr.Attributes = 2;</pre> <p>There are nine attributes that can be retrieved and five attributes that can be set. A full description of all the attribute values can be found in the help file for the FileSystemObject found on the MSDN Web site.</p>
DateCreated	<p>This read-only property returns the date and time that the specified file or folder was created. To retrieve the data and time that a folder or file was created, use the following InstallScript statement:</p> <pre data-bbox="601 1086 1005 1109">szDateTime = fldr.DateCreated;</pre> <p>The property's value is returned as a string.</p>
DateLastAccessed	<p>This read-only property returns the date and time that the specified file or folder was last accessed. To retrieve the data and time that a folder or file was last accessed, use an InstallScript statement similar to the:</p> <pre data-bbox="601 1358 1072 1381">szDateTime = fldr.DateLastAccessed;</pre> <p>The property's value is returned as a string.</p>

Table 9-12: Folder and File Object Properties (Continued)

Property	Description
DateLastModified	<p>This read-only property returns the date and time that the specified file or folder was last modified. To retrieve the data and time that a folder or file was last modified, use an InstallScript statement similar to the following:</p> <pre data-bbox="601 539 1072 560">szDateTime = fldr.DateLastModified;</pre> <p>The property's value is returned as a string.</p>
Drive	<p>This read-only property returns the drive letter of the drive on which the specified file or folder resides. To retrieve the drive on which a folder or file resides, use an InstallScript statement similar to the following:</p> <pre data-bbox="601 809 882 830">szDrive = fldr.Drive;</pre> <p>The property's value is returned as a string that contains the drive letter with a colon (:).</p>
IsRootFolder (Folder object only)	<p>This read-only property returns TRUE if the specified folder is a root folder and FALSE if the specified folder is not a root folder. To find out if a folder is a root folder, use an InstallScript statement similar to the following:</p> <pre data-bbox="601 1153 1033 1174">bRootFolder = fldr.IsRootFolder;</pre>
Name	<p>This read/write property sets or returns the name of a specified file or folder. To retrieve the name of a folder, use an InstallScript statement similar to the following:</p> <pre data-bbox="601 1356 936 1377">szFolderName = fldr.Name;</pre> <p>To set the name of a folder, use an InstallScript statement similar to the following:</p> <pre data-bbox="601 1483 936 1504">fldr.Name = szFolderName;</pre>

Table 9-12: Folder and File Object Properties (Continued)

Property	Description
Name	<p>This read/write property sets or returns the name of a specified file or folder. To retrieve the name of a folder, use an InstallScript statement similar to the following:</p> <pre data-bbox="601 508 936 529">szFolderName = fldr.Name;</pre> <p>This property does not return the complete path, only the name of the folder or file. To set the name of a folder, use an InstallScript statement similar to the following:</p> <pre data-bbox="601 707 936 728">fldr.Name = szFolderName;</pre> <p>If you try to set the name of a folder that is protected by the operating system, an exception occurs.</p>
Path	<p>This read-only property returns the path for a specified file or folder. To retrieve the path of a folder, use an InstallScript statement similar to the following:</p> <pre data-bbox="601 977 853 998">szPath = fldr.Path;</pre> <p>For drive letters, the root drive is not included. This means that the path for the C drive is C: and not C:\.</p>
ShortName	<p>This read-only property returns the short name used by programs that require the 8.3 folder and file naming convention. To retrieve the short name of a folder, use an InstallScript statement similar to the following:</p> <pre data-bbox="601 1342 991 1363">szShortName = fldr.ShortName;</pre> <p>This short name does not include the complete path of the folder or file.</p>

Table 9-12: Folder and File Object Properties (Continued)

Property	Description
ShortPath	<p>This read-only property returns the short path name used by programs that require the 8.3 folder and file naming convention. To retrieve the short path name of a folder, use an InstallScript statement similar to the following:</p> <pre data-bbox="601 539 991 560">szShortPath = fldr.ShortPath;</pre>
Size	<p>This read-only property returns the size in bytes for a folder or file. For folders, this property returns the size of all files and subfolders contained in the folder. To retrieve the size of a folder, use an InstallScript statement similar to the following:</p> <pre data-bbox="601 777 843 799">iSize = fldr.Size;</pre> <p>When using this property, remember that the InstallScript NUMBER data type is limited in the size that it can represent.</p>
Type	<p>This read-only property returns information about the type of a file or folder. To retrieve the type of a folder, use an InstallScript statement similar to the following:</p> <pre data-bbox="601 1086 855 1107">szType = fldr.Type;</pre> <p>For folders, the type that is returned by this property is the string "File Folder". For files, the type returned is based on the description associated with the registered extension.</p>

There are three methods exposed by the Folder and File objects and these are discussed in Table 9-13. All three methods apply to both types of objects. These methods are used to perform the most common kind of manipulation on folders and files; copy, move, and delete.

Table 9-13: Folder and File Object Methods

Method	Description
Copy	<p>This method copies a specified file or folder from one location to another. To copy a folder, use an InstallScript statement similar to the following:</p> <pre>fldr.Copy(destination, [overwrite]);</pre> <p>The <i>destination</i> argument specifies the location where the folder or file is to be copied. This argument needs to include the name of the folder or file to be set in the new location. The destination argument can be a relative or absolute path. If it is a relative path, it is relative to the current directory. The <i>overwrite</i> argument is optional and is used to allow the replacement of an existing folder or file. The default for this argument is TRUE.</p> <p>This method performs the same action as the CopyFile or CopyFolder methods of the FileSystemObject object except here you can only copy one folder or file. The FileSystemObject object methods can copy multiple folders and files.</p>
Delete	<p>This method deletes the specified folder or file. To delete a folder, use an InstallScript statement similar to the following:</p> <pre>fldr.Delete(force);</pre> <p>The <i>force</i> argument is optional and allows you to specify that read-only files or folders should be deleted. The default for this argument is FALSE.</p> <p>This method performs the same action as the DeleteFile or DeleteFolder methods of the FileSystemObject object except here you can only delete one folder or file. The FileSystemObject object methods can use wild cards to delete multiple folders and files.</p>

Table 9-13: Folder and File Object Methods (Continued)

Method	Description
Move	<p>This method moves a specified file or folder from one location to another. To move a folder, use an InstallScript statement similar to the following:</p> <pre data-bbox="548 469 858 497"><code>fldr.Move(destination);</code></pre> <p>The <i>destination</i> argument specifies the location where the folder or file is to be moved. This argument needs to include the name of the folder or file to be set in the new location. The destination argument can be a relative or absolute path. If it is a relative path, it is relative to the current directory. It should be noted that you cannot use wild cards as part of the <i>destination</i> argument.</p> <p>This method performs the same action as the CopyFile or CopyFolder methods of the FileSystemObject object except here you can only copy one folder or file. The FileSystemObject object methods can copy multiple folders and files.</p>

It is important to realize the difference in the three methods described in Table 9-13 and the similar methods described in Table 9-10 for the FileSystemObject object. The methods described in Table 9-13 deal with only one folder or file, whereas the methods described for copying deleting and moving folders and files in Table 9-10 can handle multiple folders and files. There are several functions in InstallScript that are similar to the methods just described. However, they are more closely similar to the methods exposed by the FileSystemObject object because they can handle multiple folders or files.

The example shown in Figure 9-10 uses some of the properties and methods that are discussed in Table 9-13. Since the value for the ProductCode property that is used in this example may be different from the one that is installed on your system you will need to check this by using Orca to open up the .msi file for InstallShield Developer.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates several of the
//                properties and methods exposed by the
//                Folder and File objects.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION      "Feedback"
#define TEMPSPEC     "C:\\Temp"
#define FOLDERSPEC   "C:\\WINNT\\Installer"
#define PRODUCTCODE "{0F031DEC-3150-4503-9744-9431264BBA48}"
#define ICONFILE     "ARPPRODUCTICON.exe"

INT      i, iAttribute(9), iAttributeValue, iAttrib;
STRING   szFolderName, szFileSpec, szDateTime, szDisplay;
STRING   szDrive, szType, szAttribDesc(9), szAttribDisplay;
OBJECT   fso, fldr, f, tempfldr;

program

    // Initialize the attributes array.
    iAttribute(0) = 0x00000000; iAttribute(1) = 0x00000001;
    iAttribute(2) = 0x00000002; iAttribute(3) = 0x00000004;
    iAttribute(4) = 0x00000008; iAttribute(5) = 0x00000010;
    iAttribute(6) = 0x00000020; iAttribute(7) = 0x00000040;
    iAttribute(8) = 0x00000080;

    // Initialize the attribute description array.
    szAttribDesc(0) = "No attributes are set";
    szAttribDesc(1) = "Read-only attribute is set";
    szAttribDesc(2) = "Hidden attribute is set";
    szAttribDesc(3) = "System attribute is set";
    szAttribDesc(4) = "Disk drive volume label is defined";
    szAttribDesc(5) = "The item is a folder";
    szAttribDesc(6) = "Archive attribute is set";
    szAttribDesc(7) = "The item is a shortcut";
    szAttribDesc(8) = "The item is compressed";

```

Figure 9-10: *Setup.rul* that demonstrates properties and methods of the Folder and File objects.


```

try
    // Create a FileSystemObject object.
    set fso = CreateObject("Scripting.FileSystemObject");

    // Create a Folder object for an existing folder.
    set fldr = fso.GetFolder(FOLDERSPEC);

    // Determine the attributes for the Folder object.
    iAttributeValue = fldr.Attributes;
    for i=0 to 8
        iAttrib = iAttributeValue & iAttribute(i);
        if(iAttrib) then
            szAttribDisplay = szAttribDisplay +
                               szAttribDesc(i) + "\n";
        endif;
    endfor;

    // Display the folder attribute information.
    sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
               "Folder Attributes\n\n%s", szAttribDisplay);

    // Create a Folder object for a uniquely named folder
    // that is to be created every time.
    set tempfldr = fso.CreateFolder(TEMPSPEC ^
                                    fso.GetTempName());

    // Display some of the folder properties.
    sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
               "Temporary Folder Properties\n\nDate created: %s\n" +
               "Folder path: %s\nFolder type: %s",
               tempfldr.DateCreated, tempfldr.Path,
               tempfldr.Type);

    // Create File object.
    set f = fso.GetFile(FOLDERSPEC ^ PRODUCTCODE ^ ICONFILE);

    // Determine the attributes for the File object.
    szAttribDisplay = "";
    iAttributeValue = f.Attributes;
    for i=0 to 8
        iAttrib = iAttributeValue & iAttribute(i);
        if(iAttrib) then
            szAttribDisplay = szAttribDisplay +
                               szAttribDesc(i) + "\n";
        endif;
    endfor;

```

Figure 9-10: *Continued.*

```

// Display the file attribute information.
sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
           "File Attributes\n\n%s", szAttribDisplay);

// Copy the file to the temporary folder.
f.Copy(tempfldr.Path ^ ICONFILE, TRUE);

// Check to see if file copy was successful.
if(fso.FileExists(tempfldr.Path ^ ICONFILE)) then
    sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
              "File copy successful.");
else
    sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
              "File copy unsuccessful.");
endif;

catch
    sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
              "Exception thrown.");
endcatch;

set f = NOTHING;
set tempfldr = NOTHING;
set fldr = NOTHING;
set fso = NOTHING;

endprogram

```

Figure 9-10: *Continued.*

This example program works with some of the folders and a file that are created when InstallShield Developer is installed. The program creates two arrays for working with the attributes of a folder or a file. The first array is an integer array that holds the individual values for each of the nine attributes that can be returned by the Attributes property. The second array holds the strings that describe the attributes. The attribute value that is returned from the Attributes property is a combined set of values that have been OR'd together. The program uses the familiar bitwise AND operator to take apart this value to see what attributes it consists of.

The example also creates a uniquely named folder under the Temp folder on drive C. It does this by using the GetTempName method of the FileSystemObject object and concatenating this unique folder name with the Temp folder location. Every time that this example is run, a new folder is created under the Temp folder. The program uses this temporary folder as the destination for copying the icon file that is added

under the Installer folder in the Windows directory. The program also accesses a few of the properties of this temporary folder and displays them in a message box.

Up to now you have learned how to create folders but to access only existing files. The next section discusses how to create text files.

TextStream Object Properties and Methods

This section examines how to create text files using the properties and methods available from the TextStream object.

A TextStream object can be created in InstallScript as follows:

```
set fso = CreateObject("Scripting.FileSystemObject");
set tso = fso.CreateTextFile("C:\\Temp\\Error.log");
```

When you create a text file, the folder in which it is to be created must already exist. If it does not, you need to create a Folder object and create the folder before creating the text file. You can also create text files using InstallScript built-in functions and you can also create binary files, something that the FileSystemObject object cannot do.

The TextStream object has four properties and nine methods that are discussed in Table 9-14 and Table 9-15. There are no InstallScript functions that duplicate the functionality provided by the properties described in Table 9-14.

Table 9-14: The TextStream Object Properties

Property	Description
AtEndOfLine	<p>This read-only property returns TRUE if the file pointer is positioned immediately before the end-of-line marker; otherwise, it returns FALSE. To find out if the program is at the end of a line, use an InstallScript statement similar to the following:</p> <pre>bEnd = tso.AtEndOfLine;</pre> <p>The file has to be opened for reading only; otherwise, an exception is thrown when using this property. There is no similar functionality in InstallScript.</p>

Table 9-14: The TextStream Object Properties

AtEndOfStream	<p>This read-only property returns TRUE if the file pointer is positioned immediately before the end-of-file marker; otherwise, it returns FALSE. To find out if the program is at the end of a file, use an InstallScript statement similar to the following:</p> <pre data-bbox="601 472 936 495">bEnd = tso.AtEndOfStream;</pre> <p>The file has to be opened for reading only; otherwise, an exception is thrown when using this property. There is no similar functionality in InstallScript.</p>
Column	<p>This read-only property returns the column number of the current character position in a text file. To find out the column of the current character, use an InstallScript statement similar to the following:</p> <pre data-bbox="601 811 839 834">iCol = tso.Column;</pre> <p>The file can be opened for reading, writing, or appending when using this property. There is no similar functionality in InstallScript.</p>
Line	<p>This read-only property returns the current line number in a text file. To find out the current line number, use an InstallScript statement similar to the following:</p> <pre data-bbox="601 1151 825 1174">iLine = tso.Line;</pre> <p>The file can be opened for reading, writing, or appending when using this property. There is no similar functionality in InstallScript.</p>

Table 9-15 describes the methods that are exposed by the TextStream object. Of the nine methods available with the TextStream object there are five InstallScript functions that perform that same or similar actions.

Table 9-15: The TextStream Object Methods

Method	Description
Close	<p>This method closes an open text file object. To close an open text file object, use an InstallScript statement similar to the following:</p> <pre>tso.Close();</pre> <p>You should always close these objects when no longer reading or writing to the file. This method is the same as InstallScript's <code>CloseFile</code> function.</p>
Read	<p>This method reads a specified number of characters from a text file and returns the resulting string. To read characters from a text file object, use an InstallScript statement similar to the following:</p> <pre>szChars = tso.Read(numchars);</pre> <p>The <i>numchars</i> argument is a value that specifies the number of characters to be read. This method is the similar to InstallShield's <code>ReadBytes</code> function.</p>
ReadAll	<p>This method reads an entire text file and returns the resulting string. To read an entire text file object, use an InstallScript statement similar to the following:</p> <pre>szChars = tso.ReadAll();</pre> <p>There is no similar function in InstallScript.</p>
ReadLine	<p>This method reads an entire line from a text file and returns the resulting string. The line returned does not include the newline character. To read an entire line from a text file object, use an InstallScript statement similar to the following :</p> <pre>szChars = tso.ReadLine();</pre> <p>The first line in a text file is line number 1. This method is the same as InstallScript's <code>GetLine</code> function.</p>

Table 9-15: The TextStream Object Methods (Continued)

Method	Description
Skip	<p>This method skips a specified number of characters when reading a text file. To skip characters when reading a text file object, use an InstallScript statement similar to the following:</p> <pre>tso.Skip(<i>numchars</i>);</pre> <p>The <i>numchars</i> argument specifies the number of characters to skip. The characters that are skipped are discarded but this does not mean that they have been removed from the text file that you are reading. This method is similar to InstallScript's <code>SeekBytes</code> function.</p>
SkipLine	<p>This method skips a line when reading a text file. To skip a line when reading a text file object, use an InstallScript statement similar to the following:</p> <pre>tso.SkipLine();</pre> <p>To skip multiple lines in a text file you would need to call this method multiple times inside a loop. There is no similar function in InstallScript.</p>
Write	<p>This method writes a string to a text file. The line that is written is not terminated with the newline character. To write a string to a text file object, use an InstallScript statement similar to the following:</p> <pre>tso.Write(<i>string</i>);</pre> <p>The <i>string</i> argument specifies the string that is to be written to the text file. This method is good for writing to a text file one continuous stream of characters. The resulting file would appear to contain just one line of text. There is no similar function in InstallScript.</p>

Table 9-15: The TextStream Object Methods (Continued)

Method	Description
WriteBlankLines	<p>This method writes a specified number of newline characters to a text file. To write a blank line to a text file object, use an InstallScript statement similar to the following:</p> <pre>tso.WriteBlankLines(<i>numlines</i>);</pre> <p>The <i>numlines</i> argument specifies the number of newline characters that are to be written to the text file. There is no similar function in InstallScript. Blank lines can be written to a text file using the WriteLine InstallScript function if you just pass the string "\r\n". This sequence is a carriage return followed by a new line.</p>
WriteLine	<p>This method writes a string to a text file. The line that is written is terminated with the newline character. To write a string to a text file object, use an InstallScript statement similar to the following:</p> <pre>tso.WriteLine(<i>string</i>);</pre> <p>The <i>string</i> argument specifies the string that is to be written to the text file. If the <i>string</i> argument is omitted then only a new line character is written to the file. This method is the same as InstallScript's WriteLine function.</p>

The sample program, shown in Figure 9-11, uses some of the methods for working with a text file that are described in Table 9-15. First, the program creates a number of objects. After creating a FileSystemObject, it creates a Folder object that has a unique name. Before the program can create the text file in a unique location, it must create the unique folder. If you try to create a text file in a nonexistent location, an exception occurs. The program creates a uniquely named folder under the Temp directory on drive C.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates several of the
//                methods exposed by the TextStream object.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION      "Feedback"
#define TEMPSPEC     "C:\\Temp"

INT      i, iCol, iLine;
STRING   szFileName, szFolder, szErrorStr, szErrorDisplay;
VARIANT Value;
OBJECT   fso, tso, fldr;

program

    try
        // Create a FileSystemObject object.
        set fso = CreateObject("Scripting.FileSystemObject");

        // Create Folder object.
        set fldr = fso.CreateFolder(TEMPSPEC ^ fso.GetTempName());
        szFolder = fldr.Path;
        szFileName = szFolder ^ "Error.log";

        // Create a TextStream object.
        set tso = fso.CreateTextFile(szFileName);

        // Add lines to the text file.
        for i=0 to 9
            Value = i + 1;
            szErrorStr = "Error Number " + Value;
            tso.WriteLine(szErrorStr);
        endfor;

        // Close the text file and
        // reopen for reading.
        tso.Close();
        set tso = NOTHING;
        set tso = fso.OpenTextFile(szFileName, 1);
    
```

Figure 9-11: *Setup.rul showing how to work with a text file.*


```

    // Read every even numbered line.
    for i=0 to 9
        if(!((i+1)%2)) then
            szErrorDisplay = szErrorDisplay +
                tso.ReadLine() + "\n";
        else
            tso.SkipLine();
        endif;
    endfor;

    // Display the error log contents.
    sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
        "Error Log Lines\n\n%s", szErrorDisplay);

    // Close the text file.
    tso.Close();

catch
    sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
        "Exception thrown.");
endcatch;

// Set the objects to NULL.
set tso = NOTHING;
set fldr = NOTHING;
set fso = NOTHING;

endprogram

```

Figure 9-11: *Continued.*

Once you have defined the absolute path for the text file, you can create this file. When you create this file, it is opened for writing so you can immediately add lines to it. The program adds 10 lines to this file inside of a loop. Once the file is complete, the program reads back the even-numbered lines from this file. To do this, the file must be closed and reopened for reading. Since you use the same OBJECT variable when you open the file for reading, you should destroy the original object before you reopen the file so the program does not consume memory unnecessarily.

The Collection Objects

You can think of a collection as an array of objects. There are three collection objects that can be created from the FileSystemObject: the Drives, Folders, and Files

collections. The Files and Drives collections expose only two properties and the Folders collection exposes the same two properties and one method.

To create a Drives collection in InstallScript, use code similar to the following.

```
set fso = CreateObject("Scripting.FileSystemObject");
set dc = fso.Drives;
```

To create a Folders collection in InstallScript, use code similar to the following:

```
set fso = CreateObject("Scripting.FileSystemObject");
szWinDir = fso.GetSpecialFolder(0);
set fldr = fso.GetFolder(szWinDir ^ "Installer");
set fldrc = fldr.SubFolders;
```

This code gets the name of an existing folder and then it creates the Folders collection by getting all the subfolders under this existing folder. To create a Files collection, use code similar to the following:

```
set fso = CreateObject("Scripting.FileSystemObject");
szWinDir = fso.GetSpecialFolder(0);
set fldr = fso.GetFolder(szWinDir ^ "Installer");
set fc = fldr.Files;
```

This code snippet gets an existing folder but, this time, gets all the files that exist under this folder.

Table 9-16 discusses the properties that are applicable to the collection objects, as well as the one method that is applicable to the Folders collection.

Table 9-16: The Collection Object Properties and Methods

Property/Method	Description
Count (Property that is applicable to all collection objects)	<p>This read-only property returns the number of items in a collection object. To obtain the number of items in a collection, use an InstallScript statement similar to the following:</p> <pre>iCnt = fldrsc.Count;</pre> <p>This same property is available with the Dictionary object.</p>

Table 9-16: The Collection Object Properties and Methods (Continued)

Item (Property that is applicable to all collection objects)	This read-only property returns an item in a collection based on a specified key. To obtain an item in a collection, use an InstallScript statement similar to the following: <code>szItem = fldrsc.Item(key);</code>
	This same property is available with the Dictionary object, but with this object the property is read/write.
Add (Method that is applicable only to a Folders collection)	This method adds a new folder to a Folders collection. To add a new folder to a Folders collection, use an InstallScript statement similar to the following: <code>fldrsc.Add(foldername);</code>
	The <i>foldername</i> argument is the name of the folder to be added to the collection. This argument is not the absolute path of the new folder.

Because InstallScript does not provide a `For...Each Next` statement, traversing collections is difficult. This is because the `Drives`, `Folders`, and `Files` collections do not have numerical indexing, so a `for` loop does not work to move through the members of a collection. To access the member of a collection, you have to know the value of the collection member.

It is possible to extend the functionality of InstallScript with a DLL and to create the capability to perform the equivalent of a `For...Each Next` statement. To create a DLL that can perform this type of action requires knowledge of COM programming using Microsoft's Active Template Library.

Figure 9-12 shows how to access the items in a collection when you know the item that you are looking for. This example is extended in the section dealing with the `Drives` collection object to look at how to traverse a `Drives` collection for the target machine.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates how to work
//                with Drives, Folders, and Files collections.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION          "Feedback"
#define PRODUCTCODE     "{0F031DEC-3150-4503-9744-9431264BBA48}"
#define ICONFILENAME    "ARPPRODUCTICON.exe"

VARIANT Alphabet(26);
OBJECT fso, fldr, fldrc, f, fc, dc, d, subfldr;

program

    // Initialize a VARIANT array with the letters of the alphabet.
    Alphabet(0) = "A"; Alphabet(1) = "B"; Alphabet(2) = "C";
    Alphabet(3) = "D"; Alphabet(4) = "E"; Alphabet(5) = "F";
    Alphabet(6) = "G"; Alphabet(7) = "H"; Alphabet(8) = "I";
    Alphabet(9) = "J"; Alphabet(10) = "K"; Alphabet(11) = "L";
    Alphabet(12) = "M"; Alphabet(13) = "N"; Alphabet(14) = "O";
    Alphabet(15) = "P"; Alphabet(16) = "Q"; Alphabet(17) = "R";
    Alphabet(18) = "S"; Alphabet(19) = "T"; Alphabet(20) = "U";
    Alphabet(21) = "V"; Alphabet(22) = "W"; Alphabet(23) = "X";
    Alphabet(24) = "Y"; Alphabet(25) = "Z";

    try
        // Create a FileSystemObject object.
        set fso = CreateObject("Scripting.FileSystemObject");

        // Create a Drives collection.
        set dc = fso.Drives;

        // Returns the Drive object for the C drive.
        set d = dc.Item(Alphabet(2));

        // Create a Folder object for the Installer folder.
        set fldr = fso.GetFolder(fso.GetSpecialFolder(0) ^
                                "Installer");

```

Figure 9-12: *Setup.rul demonstrating how to access items in a collection.*

```

        // Create a Folders collection of all subfolders.
        set fldrc = fldr.SubFolders;

        // Returns the folder object for InstallShield Developer.
        set subfldr = fldrc.Item(PRODUCTCODE);

        // Create a Files collection.
        set fc = subfldr.Files;

        // Return File object for the icon file
        // used by InstallShield Developer.
        set f = fc.Item(ICONFILENAME);

        // Display the error log contents.
        sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
            "Drive: %s\nFolder: %s\nFile: %s",
                d.Path, subfldr.Path, f.Path);

    catch
        sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
            "Exception thrown.");
    endcatch;

    // Set the objects to NULL.
    set fldr = NOTHING;
    set subfldr = NOTHING;
    set fldrc = NOTHING;
    set fc = NOTHING;
    set dc = NOTHING;
    set d = NOTHING;
    set f = NOTHING;
    set fso = NOTHING;

endprogram

```

Figure 9-12: *Continued.*

First, this program sets up a VARIANT array that holds the letters of the alphabet. The program uses the array elements to query the Drives object to return the object for a particular drive letter. As with all the other examples in this section, the example first creates a FileSystemObject. Using this object, you can directly create a Drives collection and then, using the Item property, return a Drive object. If you wanted to, you could return just the path to the drive, instead of returning the drive object. You could do this as follows:

```
szPath = d.Item(Alphabet(2));
```

Using a statement like this does not return the object itself but just the path to the Drive object referenced by the Item property. A statement such as this returns the path for a Folder or a File object, but not the object itself.

To create a Folders collection, you must first create a Folder object and then use the SubFolders property. After obtaining a Folder object from the Folders collection, you get a Files collection for the files that exist under that folder. From the Files collection, you retrieve a File object for the icon file used by InstallShield Developer. Finally, you can display the absolute path for the Drive, Folder, and File objects using the Path property that is common to all of these objects.

This example allows you to work with a collection because you knew the element that you were after. If, however, you did not know in advance the elements in a collection, you would not be able to use it. Trying to access a nonexistent element generates an exception. This provides some capability in searching for a file or a folder to see if it exists in a collection. If it does exist, the program gets the object for the folder or file. Otherwise, the program dumps into the catch block where you can take whatever action is necessary due to the absence of what you are looking for.

The Dictionary Object

A Dictionary object is an associative array where each entry is made up of a key and an item. Items can be any form of data and are stored in the array. Each item is associated with a unique key. The key is used to retrieve an individual item and is either an integer or a string. A Dictionary object is implemented using a *hash table*. The use of a hash table provides fast and consistent access time to the items in a dictionary, regardless of how large it is.

To create a Dictionary object in InstallScript, use a statement similar to the following:

```
dicto = CreateObject("Scripting.Dictionary");
```

A Dictionary object exposes four properties and six methods. The properties for the Dictionary object are discussed in Table 9-17. The InstallScript language does not contain a construct or data type similar to what you can get with the Dictionary object.

Table 9-17: Dictionary Object Properties

Property	Description
CompareMode	<p>This read/write property sets or returns the comparison mode for comparing string keys. To get the comparison mode, use an InstallScript statement similar to the following:</p> <pre>iMode = dicto.CompareMode;</pre> <p>To set the comparison mode in InstallScript, use a statement similar to the following:</p> <pre>dicto.CompareMode = 1;</pre> <p>This statement indicates that the comparison should be case insensitive when adding new values to a Dictionary object. The default is case sensitive and this is a comparison mode of 0, which is the default. If you try to change the comparison mode of a dictionary object that already contains data you will generate an error.</p>
Count	<p>This read-only property returns the number of elements in a Dictionary object. This is the same property that is available with all collection objects. To get the number of elements in this type of object, use an InstallScript statement similar to the following:</p> <pre>iCnt = dicto.Count;</pre> <p>Unlike with the collection objects discussed earlier, you can traverse a Dictionary object using a <code>for</code> loop. This is possible when you use the <code>Items</code> method and the <code>Keys</code> method to convert the Dictionary object into two separate arrays. Once you have the arrays then you can use a numerical index on these arrays and traverse the array up to an index that is one less than the value of the <code>Count</code> property.</p>

Table 9-17: Dictionary Object Properties (Continued)

Property	Description
Item	<p>This read/write property retrieves or sets the value of an item in a Dictionary object. To set the value of an element in this type of object, use an InstallScript statement similar to the following:</p> <pre data-bbox="601 504 951 530">dicto.Item(key) = newitem;</pre> <p>The <i>key</i> argument is the key that is associated with the item being set. If the <i>key</i> argument does not exist, a new element is added to the Dictionary object. If <i>newitem</i> is not set then an item that is empty is created in the Dictionary object.</p> <p>To get the value of an element in this type of object, use an InstallScript statement similar to the following:</p> <pre data-bbox="601 844 922 871">Value = dicto.Item(key);</pre> <p>If <i>key</i> does not exist when trying to retrieve an item then a new key is created with an empty item.</p>
Key	<p>This write-only property is used to change an existing key to a different key. To set a new key for an existing key in InstallScript, use a statement similar to the following:</p> <pre data-bbox="601 1151 922 1178">dicto.Key(key) = newkey;</pre> <p>The <i>key</i> argument is the key value that is being changed and the <i>newkey</i> argument is the value that is to be used for the changed key. If the value of the <i>key</i> argument is not found, a new entry in the Dictionary object is made where the item is left empty.</p>

Table 9-18 describes the methods that are available from a dictionary object. These methods are used to add key and item pairs to the Dictionary object as well as to remove these pairs from the object.

Table 9-18: Dictionary Object Methods

Method	Description
Add	<p>This method adds a key and item pair. To add a new entry in InstallScript, use a statement similar to the following:</p> <pre>dicto.Add(key, item);</pre> <p>The <i>key</i> argument identifies the key to be used and the <i>item</i> argument identifies the item value.</p>
Exists	<p>This method returns TRUE if a specified key exists; otherwise, it returns FALSE. To check if a key exists in InstallScript, use a statement similar to the following:</p> <pre>bExists = dicto.Exists(key);</pre> <p>The <i>key</i> argument specifies the key value for which you are checking the existence.</p>
Items	<p>This method returns an array that contains all the items in the Dictionary object. To obtain an array of items in InstallScript, use a statement similar to the following:</p> <pre>Array = dicto.Items();</pre> <p>This makes it possible to use a numeric index to traverse the elements in a Dictionary object.</p>
Keys	<p>This method returns an array that contains all the keys in the Dictionary object. To obtain an array of keys in InstallScript, use a statement similar to the following:</p> <pre>Array = dicto.Keys();</pre> <p>This makes it possible to use a numeric index to traverse the elements in a Dictionary object.</p>

Table 9-18: Dictionary Object Methods (Continued)

Method	Description
Remove	<p>This method removes a key and item pair from the Dictionary object. To remove a key and item pair in InstallScript, use a statement similar to the following:</p> <pre>dicto.Remove (key) ;</pre> <p>The <i>key</i> argument is the value that identifies the pair that is to be removed from the object.</p>
RemoveAll	<p>This method removes all key and item pairs from the Dictionary object. To remove all the key and item pairs in InstallScript, use a statement similar to the following:</p> <pre>dicto.RemoveAll () ;</pre>

Figure 9-13 demonstrates the use of the Dictionary object. This program is a rework of the program in Figure 9-10. In place of the two arrays that were used in the program in Figure 9-10, a Dictionary object is used to hold the attribute codes and descriptions.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This demonstrates the
//                 use of the Dictionary object.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION      "Feedback"
#define FOLDERSPEC   "C:\\WINNT\\Installer"
INT      i, iKeys(), iAttributeValue, iCnt;

```

Figure 9-13: *Setup.rul demonstrating the use of the Dictionary object.*

```

STRING  szAttribDisplay;
OBJECT  fso, fldr, dicto;

program

    try
        // Create a FileSystemObject object.
        set fso = CreateObject("Scripting.FileSystemObject");

        // Create a Dictionary object.
        set dicto = CreateObject("Scripting.Dictionary");

        // Initialize the Dictionary object.
        dicto.Add(0x00000000, "No attributes are set");
        dicto.Add(0x00000001, "Read-only attribute is set");
        dicto.Add(0x00000002, "Hidden attribute is set");
        dicto.Add(0x00000004, "System attribute is set");
        dicto.Add(0x00000008, "Disk drive volume label is defined");
        dicto.Add(0x00000010, "The item is a folder");
        dicto.Add(0x00000020, "Archive attribute is set");
        dicto.Add(0x00000040, "The item is a shortcut");
        dicto.Add(0x00000080, "The item is compressed");

        // Get size of Dictionary object
        // and size arrays accordingly.
        iCnt = dicto.Count;
        Resize(iKeys, iCnt);

        // Create a Folder object for an existing folder.
        set fldr = fso.GetFolder(FOLDERSPEC);

        // Determine the attributes for the Folder object.
        iAttributeValue = fldr.Attributes;
        iKeys = dicto.Keys();

        for i=0 to iCnt-1
            if(iAttributeValue & iKeys(i)) then
                szAttribDisplay = szAttribDisplay +
                    dicto.Item(iKeys(i)) + "\n";
            endif;
        endfor;

        // Display the folder attribute information.
        SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
            "Folder Attributes\n\n%s", szAttribDisplay);
    
```

Figure 9-13: *Continued.*

```

catch
    sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
              "Exception thrown.");
endcatch;

set fldr = NOTHING;
set dicto = NOTHING;
set fso = NOTHING;

endprogram

```

Figure 9-13: *Continued.*

This program still uses an array to be able to iterate the Dictionary object. It uses the `Keys` method to convert the key values into the `iKeys` array, and then uses the value of the array element to obtain the item from the Dictionary object.

A Drives Collection Example

The final example for the `FileSystemObject` object demonstrates a method for iterating through a `Drives` collection. Because `InstallScript` does not have a `For...Each Next` construct, you have to use another means to traverse this collection. This method takes advantage of the fact that whenever you try to access a drive letter that does not exist in the collection an exception is thrown. If you place the `try...catch...endcatch` statement inside a loop, you can continue to test each letter of the alphabet for inclusion in the `Drives` collection. When an exception is thrown, the program catches it in the `catch` block. You can then increment a loop counter and try to access the next letter in the alphabet.

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script is a composite example of using
//                 a number of different properties and methods
//                 that are exposed by the Drivers collection.
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#include "ifx.h"

```

Figure 9-14: *Setup.rul showing how to iterate through a Drives collection in InstallScript.*

```

#define CAPTION      "Feedback"
#define TEMPPATH     "C:\\Temp"

INT      i, iCnt, iFound, iType;
STRING  szKeys(), szDisplay, szFolder;
VARIANT Alphabet(26);
OBJECT  fso, dc, d, dicto, typedict, fldr, tso;

program

    // Initialize a VARIANT array with the letters of the alphabet.
    Alphabet(0) = "A"; Alphabet(1) = "B"; Alphabet(2) = "C";
    Alphabet(3) = "D"; Alphabet(4) = "E"; Alphabet(5) = "F";
    Alphabet(6) = "G"; Alphabet(7) = "H"; Alphabet(8) = "I";
    Alphabet(9) = "J"; Alphabet(10) = "K"; Alphabet(11) = "L";
    Alphabet(12) = "M"; Alphabet(13) = "N"; Alphabet(14) = "O";
    Alphabet(15) = "P"; Alphabet(16) = "Q"; Alphabet(17) = "R";
    Alphabet(18) = "S"; Alphabet(19) = "T"; Alphabet(20) = "U";
    Alphabet(21) = "V"; Alphabet(22) = "W"; Alphabet(23) = "X";
    Alphabet(24) = "Y"; Alphabet(25) = "Z";

    try
        // Create a FileSystemObject object.
        set fso = CreateObject("Scripting.FileSystemObject");

        // Create a Dictionary object to hold the
        // drive types and descriptions.
        set typedict = CreateObject("Scripting.Dictionary");

        typedict.Add(0, "Unknown drive type");
        typedict.Add(1, "Removable drive type");
        typedict.Add(2, "Fixed drive type");
        typedict.Add(3, "Network drive type");
        typedict.Add(4, "CD-ROM drive type");
        typedict.Add(5, "RAM disk drive type");

        // Create a Drives collection.
        set dc = fso.Drives;

        // Get the number of drives in the collection.
        iCnt = dc.Count;
        Resize(szKeys, iCnt);

        // Create a Dictionary object to hold the drives
        // and descriptions that are on the target machine.
        set dicto = CreateObject("Scripting.Dictionary");

```

Figure 9-14: *Continued.*

```

// Create the Error.log file.
szFolder = TEMPPATH ^ fso.GetTempName;
set fldr = fso.CreateFolder(szFolder);
set tso = fldr.CreateTextFile("Install.log");

iFound = 0;
i = 0;

while(iFound < iCnt)
    try
        // Returns the Drive object in the collection.
        set d = dc.Item(Alphabet(i));

        // Add the existing drive to the Dictionary object.
        dicto.Add(d.Path, typedict.Item(d.DriveType));

        // Write a line to the log file.
        tso.WriteLine("Success: There is a drive " +
            Alphabet(i) + " and it is a " +
            typedict.Item(d.DriveType));

        // Increment the indices.
        iFound++;
        i++;

    catch
        // Write an error line to the log file.
        tso.WriteLine("Error: There is no drive " +
            Alphabet(i));

        i++;
    endcatch;

endwhile;

// Write a final line to the log file and close the file.
tso.WriteLine("All drives have been found");
tso.Close();

szKeys = dicto.Keys();
for i=0 to iCnt-1
    szDisplay = szDisplay + szKeys(i) + "\t" +
        dicto.Item(szKeys(i)) + "\n";
endfor;

// Display the drives and types.
SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
    "Drives and Types\n\n%s", szDisplay);

```

Figure 9-14: *Continued.*

```
    catch
        sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
                  "Exception thrown.");
    endcatch;

    // Set the objects to NULL.
    set dc = NOTHING;
    set d = NOTHING;
    set dicto = NOTHING;
    set typedict = NOTHING;
    set tso = NOTHING;
    set fso = NOTHING;

endprogram
```

Figure 9-14: *Continued.*

There are several important points that you need to note. First, the array that holds the letters of the alphabet is of type VARIANT. This is necessary because an exception is thrown if, for example, the array is typed as STRING.

The second important point is that you need to nest one try...catch...endcatch statement inside another. The nested statement is inside the while loop and the outside try...catch...endcatch statement is where you create the objects that you use in the example. As you find the drives in the Drives collection, the drive letter plus the drive type is added to a Dictionary object. With the Dictionary object, you can use the standard approach for traversing an array by converting the keys in the Dictionary object into an array. You can then use the elements in the array to access the items in the Dictionary object.

The final point to note is that the program uses a TextStream object to write a log of all the actions that are performed in this example. This shows an approach that you can use to create a log of all actions performed during an installation. Creating such a log file can be valuable when a customer has a problem installing your product.

The Windows Script Host Objects

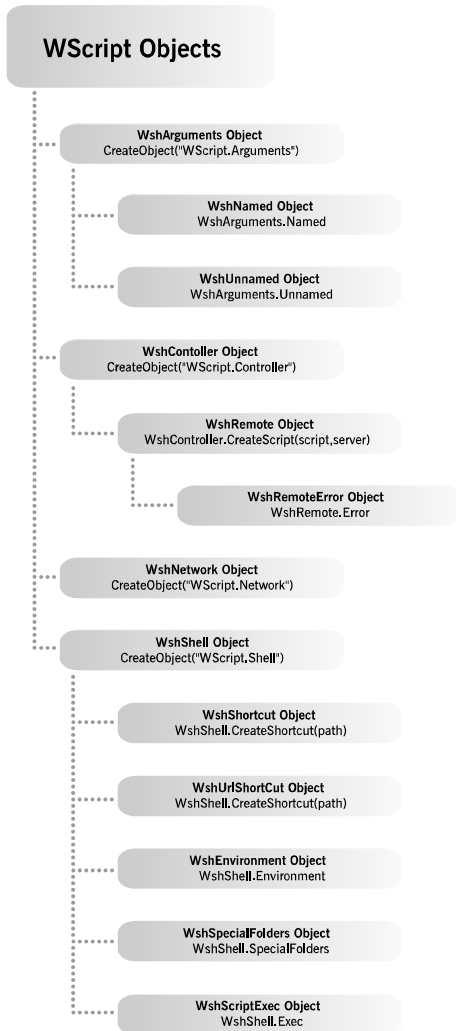


Figure 9-15: *The Windows Script Host object model.*

The Windows Script Host is an application that allows scripts to be run on 32-bit Windows platforms (Figure 9-15). It can be considered the replacement for the old MS-DOS batch file. The Windows Script Host provides a number of useful objects for performing operations during an installation. The root object in the hierarchy shown in Figure 9-15 is the WScript object. This object is not creatable, but it is always available to scripts. Since you cannot create it, you do not have access to this object from InstallScript. However, all the other objects shown in Figure 9-15 are creatable and therefore can be created in InstallScript using the `CreateObject` function.

A complete coverage of the objects that compose the Windows Script Host is outside the scope of this book. This chapter provides an introduction to this technology and the possibilities that are provided for use in your installation programs.

The Creatable Objects

This section provides an overview of the creatable objects that you can access from InstallScript. Then, it takes a closer look at some of these objects.

There are 13 Windows Script Host objects that you can create in InstallScript (Table 9-19).

Table 9-19: Windows Script Host Creatable Objects

Object	Description
WshArguments	This object is a collection of the arguments that are passed to a script. This object has no use when using InstallScript. Its only value is when running VBScript or JScript files from the command line.
WshNamed	Accessing the Named property of the WshArguments object creates this object. This object is a collection of all arguments sent to a script that have names. This object has no use when using InstallScript. Its only value is when running VBScript or JScript files from the command line.

Table 9-19: Windows Script Host Creatable Objects (Continued)

Object	Description
WshUnnamed	<p>Accessing the Unnamed property of the WshArguments object creates this object. This object is a collection of all arguments sent to a script that do not have names. This object has no use when using InstallScript. Its only value is when running VBScript or JScript files from the command line.</p>
WshContoller	<p>The sole purpose of this object is to provide access to the CreateScript method that is used to create a script process on a remote machine. You can instantiate this object using an InstallScript statement similar to the following:</p> <pre data-bbox="565 790 1216 814">set cntrl0 = CreateObject("WScript.Controller");</pre> <p>This object has only one method and no properties.</p>
WshRemote	<p>This object allows you to remotely administer computer systems on a computer network. Through this object interface, you can manipulate other programs or scripts. You can instantiate this object using an InstallScript statement similar to the following:</p> <pre data-bbox="565 1097 1202 1121">set remscript = CreateScript(script, [server]);</pre> <p>The <i>script</i> argument identifies either a VBScript or JScript file that is to be run remotely. The <i>server</i> argument identifies the remote server on which the script is to be run. This argument is optional.</p>
WshRemoteError	<p>This object provides access to the error information available when a remote script terminates as a result of a script error. The remote script is one that is created with the WshRemote object. Accessing the Error property of the WshRemote object creates this object.</p>

Table 9-19: Windows Script Host Creatable Objects (Continued)

Object	Description
WshNetwork	<p>The object provides access to the shared resources on the network to which the target computer is connected. You can instantiate this object using an InstallScript statement similar to the following:</p> <pre data-bbox="565 508 1189 530">set network = CreateObject("WScript.Network");</pre> <p>You would create a WshNetwork object when you want to connect to network shares and network printers, disconnect from network shares and network printers, map or remove network shares, or access information about a user on the network.</p>
WshShell	<p>This object provides access to the Windows shell. You can instantiate this object using an InstallScript statement similar to the following:</p> <pre data-bbox="565 883 1136 906">set shell = CreateObject("WScript.Shell");</pre> <p>The properties and methods of this object allow you to work with the system folders, shortcuts, registry, and environment variables.</p>
WshShortcut	<p>This object allows you to create a shortcut programmatically. You can instantiate this object using an InstallScript statement similar to the following:</p> <pre data-bbox="565 1188 1136 1211">set shortcut = shell.CreateShortcut(path);</pre> <p>The path argument is the absolute path to the shortcut (.lnk) file that is to be created.</p>

Table 9-19: Windows Script Host Creatable Objects (Continued)

Object	Description
WshUrlShortcut	<p>This object allows you to programmatically create a shortcut that references an Internet location. You can instantiate this object using an InstallScript statement similar to the following:</p> <pre data-bbox="565 504 1136 536">set shortcut = shell.CreateShortcut(path);</pre> <p>The <i>path</i> argument is the absolute path to the shortcut (.url) file that is to be created.</p>
WshEnvironment	<p>This object is a collection that contains all the environment variables on the target machine. You can instantiate this object using an InstallScript statement similar to the following:</p> <pre data-bbox="565 807 1093 839">set envc= shell.Environment([strType]);</pre> <p>The <i>strType</i> argument identifies the location from where the environment variable is to be read.</p>
WshSpecialFolders	<p>This object is a collection that defines the location of all the system-defined folders. You can use this object using an InstallScript statement similar to the following:</p> <pre data-bbox="565 1076 1093 1107">szFolder= shell.SpecialFolders(folder);</pre> <p>The <i>folder</i> argument is the name of the special folder for which you want the location.</p>
WshScriptExec	<p>This object provides information about a script that is run using the Exec method. You can use this object using an InstallScript statement similar to the following:</p> <pre data-bbox="565 1344 985 1375">set exec = shell.Exec(command);</pre> <p>The <i>command</i> argument specifies the name of the script along with the arguments of the script being executed.</p>

The objects available with the Windows Script Host provide a lot of functionality that can be used during an installation. The two most frequently used objects are the WshNetwork and the WshShell objects. The next two sections take a brief look at these two objects along with a few examples of using these objects in InstallScript.

The WshNetwork Object

The WshNetwork object exposes three properties and seven methods. This object is generally used to work with computers that are attached to a network, but it will work with a computer that is not connected. A WshNetwork object is created in the following way:

```
set network = CreateObject("WScript.Network");
```

The three properties exposed by this object are described in Table 9-20.

Table 9-20: WshNetwork Object Properties

Property	Description
ComputerName	This read-only property returns the name of the computer system as a string. To access this property in InstallScript, use a statement similar to the following: <code>szComputerName = network.ComputerName;</code>
UserDomain	This read-only property returns the name of the user domain as a string. To access this property in InstallScript, use a statement similar to the following: <code>szUserDomain = network.UserDomain;</code>
UserName	This read-only property returns the user name of the person signed on to the computer as a string. To access this property in InstallScript, use a statement similar to the following: <code>szUserName = network.UserName;</code>

There are eight methods that are exposed by the WshNetwork object. These methods are described in Table 9-21.

Table 9-21: WshNetwork Object Methods

Method	Description
AddPrinterConnection	This method adds a network printer to an MS-DOS printer port, such as LPT1. You cannot use this method to add a remote Windows-based printer connection.
AddWindowsPrinterConnection	<p>This method is similar to using the Printer option on Control Panel to add a printer connection. Unlike the AddPrinterConnection method, this method allows you to create a printer connection without directing it to a specific port, such as LPT1. To use this method in InstallScript, use a statement similar to the following:</p> <pre>network.AddWindowsPrinterConnection(strPrinterPath, strDriverName, [strPort]);</pre> <p>The last two arguments are ignored on Windows NT/2000/XP. The last argument is optional.</p>
EnumNetworkDrives	<p>This method returns a collection that is an array that associates pairs of items, network drive local names and their associated UNC names. An even-numbered item in the collection represents the local name of a logical drive and an odd-numbered item represents the associated UNC share name. To use this method in InstallScript, use a statement similar to the following:</p> <pre>set netdrives = network. EnumNetworkDrives;</pre>

Table 9-21: WshNetwork Object Methods (Continued)

Method	Description
EnumPrinterConnections	<p>This method returns a collection that is an array that associates pairs of items, network printer local names, and their associated UNC names. An even-numbered item in the collection represents a printer port and an odd-numbered item represents the networked printer UNC name. To use this method in InstallScript, use a statement similar to the following:</p> <pre>set printers = network. EnumPrinterConnections;</pre>
MapNetworkDrive	This method adds a shared network drive to the target computer system.
RemoveNetworkDrive	This method removes a shared network drive from the target computer system.
RemovePrinterConnection	This method removes a shared network printer connection from the target computer system.
SetDefaultPrinter	This method assigns a remote printer the role of default printer on the target system.

Figure 9-15 provides an example program that uses the properties and one method from the WshNetwork object. This simple example enumerates the printer connections that are available on the target machine. It also retrieves the values of the properties, which define the name of the computer, the name of the user signed on to the system, and domain of the logged on user. Looping through the collection of printer connections is easy since this collection does have a numeric index. This is better than what you experienced with the collections that are available with the FileSystemObject object.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates some of the
//                  properties and methods of the WshNetwork object.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION      "Feedback"

INT      i, iCnt;
STRING   szComputerName, szUserName, szUserDomain, szDisplay;
OBJECT   network, printers;

program

    try

        // Create a WshNetwork object.
        set network = CreateObject("WScript.Network");

        // Retrieve the properties of the WshNetwork object.
        szComputerName = network.ComputerName;
        szUserName = network.UserName;
        szUserDomain = network.UserDomain;

        // Enumerate the printers on the target system.
        set printers = network.EnumPrinterConnections;
        iCnt = printers.Count;

        // Loop through the printers collection and get
        // the port and printer names that were enumerated.
        for i=0 to iCnt-1 step 2
            szDisplay = szDisplay + "Port " + printers.Item(i) +
                " = " + printers.Item(i+1) + "\n";
        endfor;
    
```

Figure 9-15: *Setup.rul* for demonstrating the *WshNetwork* object properties and methods.


```

        // Display the network information.
        sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
            "Computer name: %s\nUser name: %s\nUser domain: %s\n" +
            "%s", szComputerName, szUserName,
            szUserDomain, szDisplay);

    catch
        sprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
            "Exception thrown.");
    endcatch;

    // Set the object to NULL.
    set network = NOTHING;

endprogram

```

Figure 9-15: *Continued.*

It is best to run this example on Windows NT or Windows 2000, or the properties may return an empty string for one or more of these properties. When enumerating the printer connections, the Count property will always be an even number since for each printer there are two values in the array that is returned. The array of connected printers is based on the printer drivers that have been installed. If the printer is connected locally then the port will be the same for all printers that are connected. This port is usually LPT1.

The WshShell Object

As previously mentioned, the WshShell object provides a powerful set of properties and methods that can be used to access all the facilities of the Windows shell. This includes being able to create shortcuts, create and retrieve registry entries, obtain the values of environment variables, and get the paths to special folders in Windows.

In InstallScript, a WshShell object is created with a statement similar to the following:

```
set shell = CreateObject("WScript.Shell");
```

This object has three properties and ten methods. Some of these properties and methods create other objects. This section does not extensively detail the properties and methods of these other objects. The properties of the WshShell object are discussed in Table 9-22.

Table 9-22: WshShell Object Properties

Property	Description
CurrentDirectory	<p>This read/write property retrieves or sets the current directory. To retrieve the value of the current directory in InstallScript, use a statement similar to the following:</p> <pre data-bbox="548 472 979 495">szPath = shell.CurrentDirectory;</pre> <p>To set the value of the current directory in InstallScript, use a statement similar to the following:</p> <pre data-bbox="548 605 979 627">shell.CurrentDirectory = szPath;</pre> <p>This property is similar to the <code>ChangeDirectory</code> function in InstallScript, except this function cannot be used to return the present location of the current directory.</p>
Environment	<p>This property returns the <code>WshEnvironment</code> object, which is a collection of all the environment variables on the target system. To retrieve the value of the <code>PATH</code> environment variable in InstallScript, use statements similar to the following:</p> <pre data-bbox="548 980 1100 1033">set environment = shell.Environment; szEnvironment = environment.Item("PATH");</pre> <p>This property provides a similar functionality as the <code>GetEnvVar</code> function in InstallScript.</p>
SpecialFolders	<p>This property returns a <code>WshSpecialFolders</code> object, which is a collection of special folder locations. These special folders are set by the operating system. To retrieve the value of the <code>Desktop</code> special folder location in InstallScript, use statements similar to the following:</p> <pre data-bbox="548 1351 1115 1404">set specfldrs = shell.SpecialFolders; szDesktopPath = specfldrs.Item("Desktop");</pre> <p>There is no similar function in InstallScript.</p>

The 10 methods of the WshShell object are discussed in Table 9-23.

Table 9-23: WshShell Object Methods

Method	Description
AppActivate	<p>This method activates an application window by moving the focus to the application. The application needs to be running before this method can be used. To make Notepad the top window in InstallScript, use a statement similar to the following:</p> <pre>bSuccess = shell.AppActivate("Notepad");</pre> <p>Notepad would have to already be launched and not be minimized in order to see anything happen. This method does not change whether a window is minimized or maximized. There is no similar function in InstallScript.</p>
CreateShortcut	<p>This method creates a new WshShortcut or WshUrlShortcut object. To create a shortcut to an Internet location on the desktop in InstallScript, use statements similar to the following:</p> <pre>set specfldrs = shell.SpecialFolders; szDesktopPath = specfldrs.Item("Desktop"); set shortcut = shell.CreateShortcut (szDesktopPath ^ "InstallShield.url"); shortcut.TargetPath = "http://www.installshield.com"; shortcut.Save;</pre> <p>This code creates a shortcut on the desktop that launches the InstallShield Web site. This method is similar to what can be accomplished in the AddFolderIcon function in InstallScript.</p>

Table 9-23: WshShell Object Methods (Continued)

Method	Description
ExpandEnvironmentStrings	<p>This method returns an environment variable's expanded value. To expand the value of the COMSPEC environment variable into its full value in InstallScript, use statements similar to the following:</p> <pre>set environment = shell.Environment; szPath = shell.ExpandEnvironmentStrings (environment.Item("COMSPEC"));</pre> <p>In InstallScript the GetEnvVar function expands the environment string before it returns the value.</p>
LogEvent	<p>This method adds an event entry to the application log file on Windows NT/2000 and to the WSH.log file on Windows 9x. To log an error event in InstallScript, use a statement similar to the following:</p> <pre>shell.LogEvent(1, "Error message.");</pre> <p>There is no similar function in InstallScript.</p>
Popup	<p>This method displays text in a message box that can have a different number of buttons and icons displayed.</p>
RegDelete	<p>This method deletes a key or one of its values from the registry. This method performs the same actions that the RegDBDeleteKey and RegDBDeleteValue functions do in InstallScript.</p>

Table 9-23: WshShell Object Methods (Continued)

Method	Description
RegRead	This method returns the value of a key or value-name from the registry. This method is similar to the RegDBGetKeyValueEx function in InstallScript.
RegWrite	This method creates a new key, adds another value-name to an existing key, or changes the value of an existing value-name. This method is similar to the RegDBCreateKeyEx and RegDBSetKeyValueEx functions in InstallScript.
Run	This method runs a program in a new process based on the command line that is passed as an argument. This method is similar to the InstallScript functions LaunchApp and LaunchAppAndWait.
SendKeys	This method sends one or more keystrokes to the active window as if they had been typed from the keyboard. There is no similar function in InstallScript.

Figure 9-16 provides an example program that uses some of the properties and methods of the WshShell object. In this program, a few of the properties and methods used have not been discussed in the previous two tables. Refer to the Windows Script Host documentation for the complete details of what is being done in this example.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates some of the
//                properties and methods of the WshShell object.
//
////////////////////////////////////

#include "ifx.h"

#define CAPTION      "Feedback"

INT      iButton;
BOOL     bSuccess;
STRING   szComSpecPath, szEnvironment, szDesktopPath;
STRING   szCurDir, szText, szTitle;
OBJECT   shell, specfldrs, environment, shortcut;
program

    try
        // Create a WshShell object.
        set shell = CreateObject("WScript.Shell");

        // Create a WshSpecialFolders object using
        // the SpecialFolders property.
        set specfldrs = shell.SpecialFolders;

        // Create a WshEnvironment object using
        // the Environment property.
        set environment = shell.Environment("SYSTEM");

        // Get the expanded absolute path to CMD.EXE.
        szComSpecPath = shell.ExpandEnvironmentStrings
            (environment.Item("COMSPEC"));

        // Get the location of the current directory.
        szCurDir = shell.CurrentDirectory;

        // Get the path to the Desktop folder location.
        szDesktopPath = specfldrs.Item("Desktop");

        // Print out the display of values.
        SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
            "COMSPEC: %s\nCurrent directory: %s\nDesktop: %s",
            szComSpecPath, szCurDir, szDesktopPath);

```

Figure 9-16: *Setup.rul demonstrating the properties and methods of the WshShell object.*

```

// Create message box strings.
szText = "Do you want to create a shortcut on the " +
        "desktop for the InstallShield Web site?";
szTitle = "Create URL Shortcut";

// Display the Yes/No message box.
iButton = shell.Popup(szText, 0, szTitle, 4 + 32);

// Create a shortcut on the desktop if the
// Yes button is clicked.
if(iButton = 6) then
    set shortcut = shell.CreateShortcut(szDesktopPath ^
        "InstallShield.url");
    shortcut.TargetPath = "http://www.installshield.com";
    shortcut.Save;
endif;

// Log the fact that the program completed successfully.
shell.LogEvent(0, "The program worked successfully.");
catch
// Log that the program did not complete successfully.
shell.LogEvent(1, "The program had error.");

    SprintfBox(MB_OK | MB_ICONINFORMATION, CAPTION,
        "Exception thrown.");
endcatch;

// Set the objects to NULL.
set shell = NOTHING;
set specfldrs = NOTHING;
set environment = NOTHING;
set shortcut = NOTHING;

endprogram

```

Figure 9-16: *Continued.*

The first part of this program retrieves the values of several properties and then displays the results. It then uses the `Popup` method to display a message box with Yes/No buttons that asks the end user if they want to create a shortcut on the desktop. If the end user clicks Yes, the program creates an Internet shortcut that launches the InstallShield Web site. After creating a `WshUrlShortcut` object by passing the shortcut name with an `.url` extension, the program uses the `TargetPath` property to point the shortcut at the InstallShield Web site. Before the shortcut actually is created, you have to use the `Save` method.

Finally, the example uses a method of the WshShell object that allows you to write to the Application Event Log on Windows NT/2000. The LogEvent method, when run on Windows 9x, will write to the Wsh.log file that is found in the Windows directory. To view the messages that are written to the Application Event Log, go to the Event Viewer and indicate that you want to look at the events logged for applications. The events that are logged using this method appear with WSH as the source. If you right click on the event and select Properties, a dialog appears with the message that was written to the Event Log.

More Objects

This chapter has covered three objects that can be accessed from InstallScript. These are not the only objects that are useful in the development of an installation program. In this section, you will learn about three additional objects. You will need to access the MSDN Library to obtain the documentation for these objects.

The first object is the WebBrowser control that provides the capability for browsing, document viewing, and data downloading in your installation programs. To create a browser object in InstallScript, use a statement similar to the following:

```
set ieo = CreateObject("InternetExplorer.Application");
```

In addition to the FileSystemObject and Wscript objects, the Shell object can be used to access files, folders, shortcuts, and printers. The Shell object is created in InstallScript using a statement similar to the following:

```
set shell = CreateObject(Shell.Application");
```

Access the MSDN Library for documentation on how to use the Shell object. Finally there is an object that can be used to provide database access from InstallScript. This is the object that provides access to the properties and methods of the Active Data Object (ADO). To create a connection object from which all other objects are generated in InstallScript, use a statement similar to the following:

```
set dbconnection = CreateObject("ADODB.Connection");
```

This completes the discussion of how to access COM from InstallScript and the importance of using the exception handling mechanism that is provided.

Conclusion

This chapter has provided two major ways to enhance your installation programs. The first was the discussion of how to implement exception handling in InstallScript. Exception handling is important because it enables you to keep a program from failing unnecessarily or, in the worst case, allows for a graceful way to terminate an installation program.

The second way to enhance your installation programs is to use COM to extend InstallScript's built-in functionality. With COM, you need to make use of the exception handling mechanism described above. Many exceptions are thrown when trying to access the automation interface provided by some object. The COM objects that you have access to are those that you can create using InstallScript's `CreateObject` function. You are not able to access objects that are available in the environment without first creating a DLL that implements the equivalent of the `GetObject` method that is available in Visual Basic. This chapter also examined the capabilities of three automation interfaces that can be important in installation program creation. These interfaces are those exposed by the Windows Installer engine, the Scripting Run-time, and the Windows Script Host. There are additional interfaces that might be of value. These are the interfaces exposed by the Browser control, the Windows shell, and the interface exposed by the Active Data Object.

It is important to realize that this chapter does not provide all the details necessary to use these automation interfaces to their fullest capabilities. You should obtain the documentation on each one of them in order to learn the details required to make full use of these capabilities.

Part III

Getting Down to Business

Chapter

10

Common Installation Tasks

So far in this book, you have created both a Standard and a Basic MSI project for the Developer Art application. These were very basic projects with nothing added other than a shortcut on the Start\Programs menu. This chapter discusses how to build in additional functionality in an installation project. When you add this functionality to an installation project, you do it via components. Changes to the target system are made if the associated component is installed.

This chapter examines how to create file associations, create initialization files, create empty folders, and work with environment variables. You will also learn about setting launch conditions, as well as performing searches of the target system for installed applications and particular registry entries.

Creating File Associations

A file association is where a particular file extension is associated with an extension server. A file association is also known as a file type. An extension server is normally an executable that can open a file with a certain file extension. Creating file associations is an important part of the "Certified for Windows" logo requirements as specified in the *Application Specification for Microsoft Windows 2000 for desktop applications*. This specification can be downloaded from the following MSDN Web site:

<http://msdn.microsoft.com/certification/appspec.asp>

This section shows how to create the simplest form of a file association for the Developer Art application. The Developer Art application saves files with an .idv extension. Also, even though it is not a requirement of the "Certified for Windows" logo, we will also look at the creation of a MIME type. A MIME (Multipurpose Internet Mail Extension) type defines the server that is used to open a file that is an email attachment.

Since there is no difference in how a file type is created between a Standard project and a Basic MSI project, this chapter uses the Standard project you created in Chapter 5 in its discussions of this subject. First we will look at a summary of the requirements for creating file associations as defined in the *Application Specification for Microsoft Windows 2000 for desktop applications*.

The "Certified for Windows" Logo Requirements for File Associations

The general requirement is that all non-hidden files that are either created during the installation outside the application's directory or created by the application as part of its normal function need to have an associated file type, an identifying icon, a description, and an associated action that is implemented when you double click on the file in Windows Explorer. In particular the files that need to be provided with a file association are as follows:

- Non-hidden files created during the installation outside of the main install directory of the application.

- Non-hidden implementation and data files that reside outside of the main install directory of the application.
- Non-hidden files that are user created and are native to the application.

If a file association has already been registered by another application, it is acceptable to let the previous registration stand or you can choose to take over the file association for your application. Microsoft now recommends that file extensions be four or five characters long to avoid the conflict of two different applications registering a file association for the same extension.

For every file association that you register in your installation program, you need to do the following:

- Provide an icon for each file that is registered so that none of these files uses the default Windows icon in Windows Explorer.
- Provide a good description of the file that will appear in Windows Explorer instead of the default description that consists of the extension followed by the word "File".
- Ensure that there is an appropriate action associated with the file so when an end user double clicks on the file in Windows Explorer, the application is loaded and the file is opened unless the file is designated as the "NoOpen" type in the registry.

The NoOpen designation is used for files that you do not want end users to open. When the end user double clicks a file marked as NoOpen, the operating system automatically provides a message informing the user that the file should not be opened. Note that if an action is later associated with a NoOpen file type, the NoOpen designation is ignored and the operating system attempts to open the file.

Creating a File Association for the Developer Art Application

A file association needs to reference the particular file that will serve as the extension server. This means that the file association should be defined as part of the

component that installs the extension server. In the Developer Art application, this component is DeveloperArt and this component installs the DeveloperArt.exe file, which is the key path for its component. It is in this component that you will create the file association.

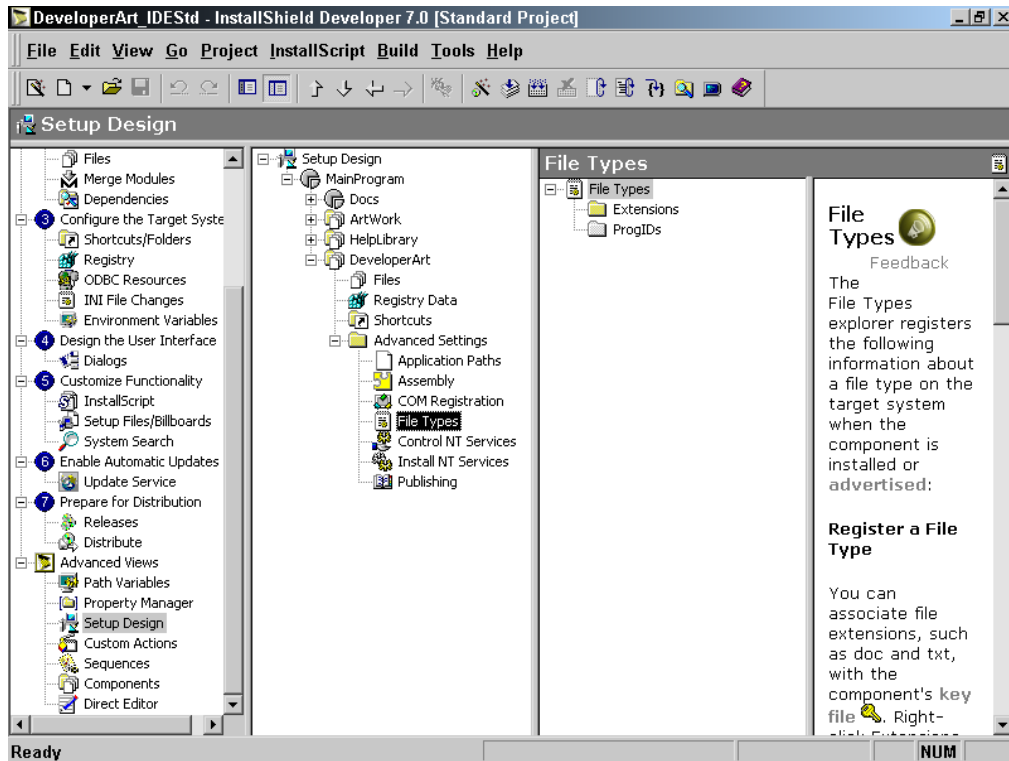


Figure 10-1: *The File Types icon under the DeveloperArt component.*

For this example and all examples in this chapter, you will use the DeveloperArt_IDEStd project. To define a file association for the Developer Art application, perform the following five steps:

1. Open the DeveloperArt_IDEStd project and go to the DeveloperArt component under Setup Design in Advanced Views. Click the File Types icon under Advanced Settings (Figure 10-1) and in the File Types panel, right-click on the Extensions icon and select New Extension.

- In the edit field that is created, enter the extension for which you are creating the file association. Do not type a period. In this example, type “idv” without quotes. Beneath this extension, the Open canonical verb is provided by default.

File associations use verbs as shorthand for actions that are invoked by the Windows shell. A canonical verb is one that can be used with any language and the operating system will generate a properly localized display string. In most cases a file association has a preferred action when an end user double-clicks on a file in Windows Explorer. The verb that is linked to this preferred action is called the primary verb. Since the Open verb is the most common primary verb, it is provided by default when you create an extension.

- When you have entered the .idv extension without the period, you need to fill in the progID property for this extension. You can use a simple format for creating the ProgID and this consists of the extension followed by the word “file.” This approach would give a ProgID equal to “idvfile” and this is the approach used for text files and the .txt extension.

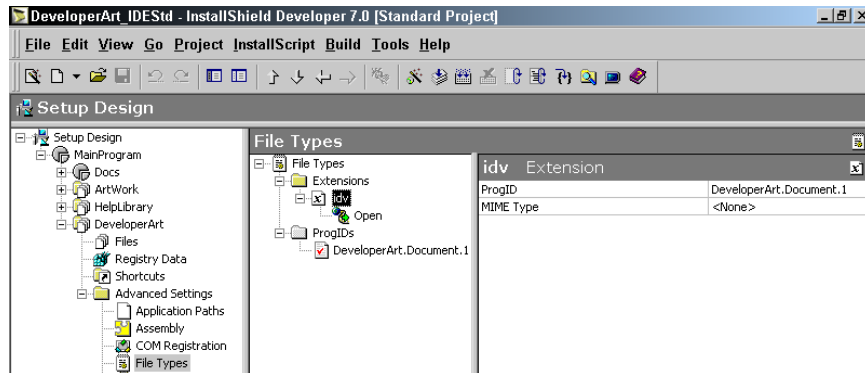


Figure 10-2: *The File Types view for the .idv extension with the ProgID specified.*

However, the proper approach is to use a format where the ProgID consists of the name of the application followed by the application element to be opened, which is followed by the version number of the application to be used. In our example you would get a ProgID equal to “DeveloperArt.Document.1” and this is what you want to use. Note that

the three parts of this ProgID are delimited by periods. The entries you make here for the extension are used to populate the Extension and ProgID tables in the database. When you enter this string into the ProgID property for the extension you get something that looks like what is shown in Figure 10-2.

4. Ignore the MIME Type property and click the Open verb to enter the properties for the Open command. All the entries that you make here relate to the context menu that appears when an end user right-clicks on a registered file name in Windows Explorer. The values that you enter here populate the Verb table in the database.

The Command Sequence property specifies the position that the Open verb will have on the context menu. This property is optional, but to ensure that the Open verb appears at the top of the context menu, give it a sequence of 0. This forces this command to appear at the top of the context menu in bold.

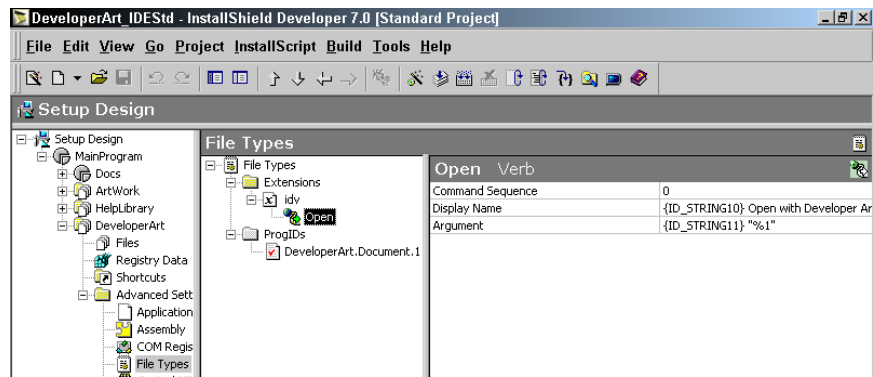


Figure 10-3: *The property values for the Open verb for the Developer.Art component.*

The Display Name property contains the name that appears on the context menu. If you leave this property NULL, the command is displayed on the context menu. Type the string “Open with Developer Art” in this field.

The Argument field is a placeholder for the file that will be passed to the extension server, which in this case is the file DeveloperArt.exe. For the Open verb the argument is normally "%1". For this example, type "%1"

with the quotes. Surround the %1 placeholder with quotes because of the possibility that the file the extension will launch has a long path name.

After making the above entries for the Open verb, you will see something like what is shown in Figure 10-3. Note that InstallShield Developer has inserted default string IDs for the Display Name and the Argument property values. This is to facilitate the localization of the installation program.

5. The last thing that you need to do to complete the entries required to define a file association is to enter the properties for the ProgID that you created earlier. Click on the ProgID that you created to display the property sheet. All the properties that you are interested in here relate to the display of the file in Windows Explorer. The entries that you make here populate a row in the ProgId table. Because the extension server is not a COM server, you can leave the COM Class property empty.

The Description property indicates what is displayed in Windows Explorer beside a file with the registered extension. If you leave this property NULL, Windows Explorer shows a description of "IDV File," which is not very informative. Type "Developer Art File" without quotes.

The Icon File property identifies a file from which an icon can be extracted and which will be used to identify any file that has an .idv extension. You can browse to any file and extract an icon for the Icon File property. For this example you should browse to the SHELL32.DLL file in the system folder. This file is a source for many interesting icons.

The Icon Index property designates the specific icon that is to be extracted from the icon file specified in the previous property. Type or select 41 for this property.

When you finish making these entries, you will see what is shown in Figure 10-4. If you wanted to designate a file as being NoOpen then you would delete the Open verb under the extension name and then you would use the Registry table to enter a value name of NoOpen under the ProgId. You would then give this value name a string as its data that would be displayed in a message box when someone double

clicked on the file in Windows Explorer. Normally this string would state that this type of file should not be opened.

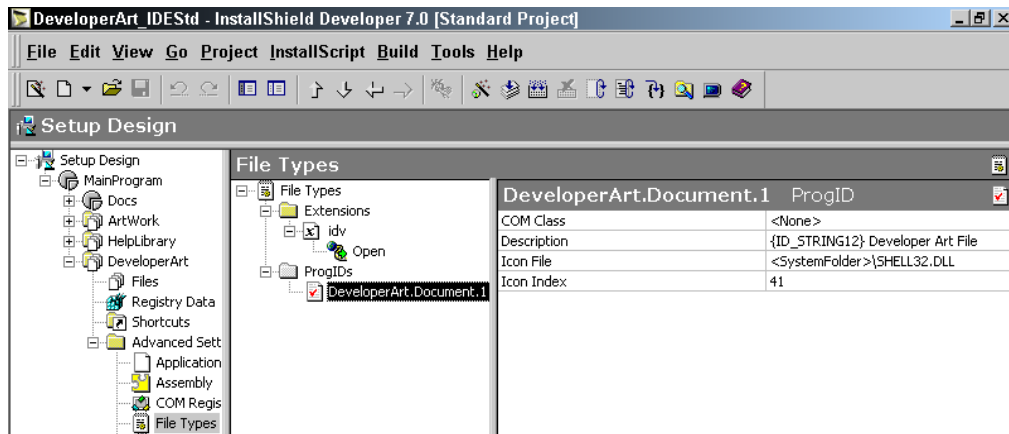


Figure 10-4: *The ProgID properties for the DeveloperArt component file association.*

The next thing that you have to do is to see what happens with the installation of the Developer Art application now that you have created a file association. To test the creation of a file association, do the following::

1. Build the project by clicking the Build button on the Toolbar.
2. Install the Developer Art application.
3. Run the Developer Art application.
4. Create and save an .idv file.
5. Examine the created file in Windows Explorer and right-click on it to display the context menu.

When you examine the file that you created with the Developer Art application in Windows Explorer, you should see something like what is shown in Figure 10-5.

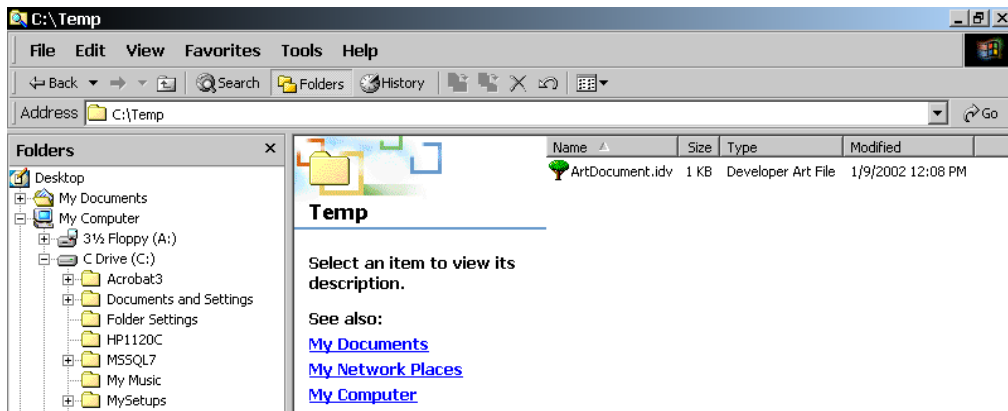


Figure 10-5: *The ArtDocument.idv file in Windows Explorer.*

When you right-click on this file in Windows Explorer, a context menu is displayed (Figure 10-6). The context menu has the string “Open with Developer Art” at the top. This string appears in bold because it is the first item on the context menu. This is a result of setting the Command Sequence property for the Open verb to 0.

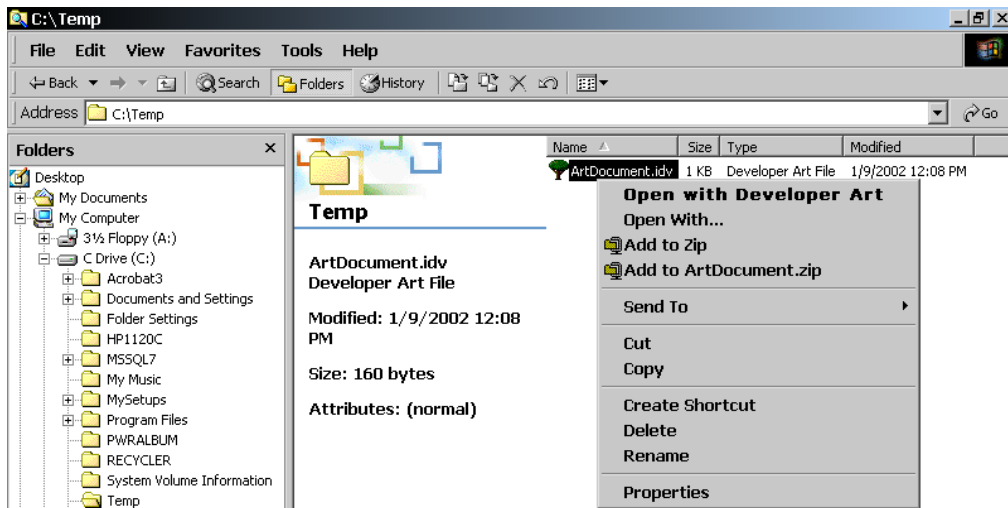


Figure 10-6: *The right-click context menu for the ArtDocument.idv file.*

To provide this functionality, the installation writes to three different locations in the registry. All three locations are under the HKEY_CLASSES_ROOT (HKCR) key.

The first sub-key under HKCR is the extension key where the name of the key is the extension that is being registered (Figure 10-7).

The default value for this key is the ProgID that is associated with the extension. There are two sub-keys under this key that are created, but in this application, these keys have no meaning since you are not allowing the creation of a new document from the context menu in Windows Explorer. When you access a file with the .idv extension, this is the first key that the shell searches for. When it finds this key, it reads the value of the ProgID and then searches for a key under HKCR that has the ProgID as its name.

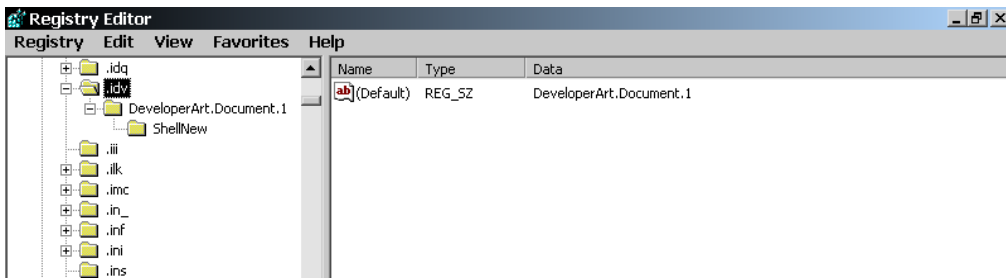


Figure 10-7: *The extension key for the Developer Art file association.*

The structure of the ProgID key under HKCR is shown in Figure 10-8. When the shell finds this key, it reads the default value for the `Shell\Open\command` key, which consists of the command line for opening the file with the extension server.

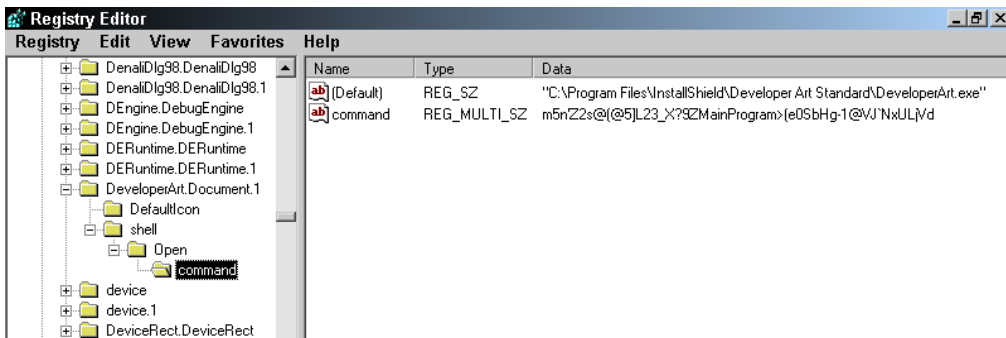


Figure 10-8: *The ProgID registry key for the Developer Art file association.*

Note that there is a value name `command` under the `Shell\Open\command` key and it has as its data what appears to be a string of garbage characters. These garbage characters make up what is termed a Darwin Descriptor. This entry in the registry is used to install an advertised application through the activation of a file that uses this application as an extension server. The Darwin Descriptor is discussed a little more in Chapter 13.

The third location to which your installation writes to the registry has to do with making the application available in the Open With dialog by which the end user can choose what program they want to use to open a specific file. The key for enabling this functionality is in the following location in the registry:

```
HKEY_CLASSES_ROOT\Applications
```

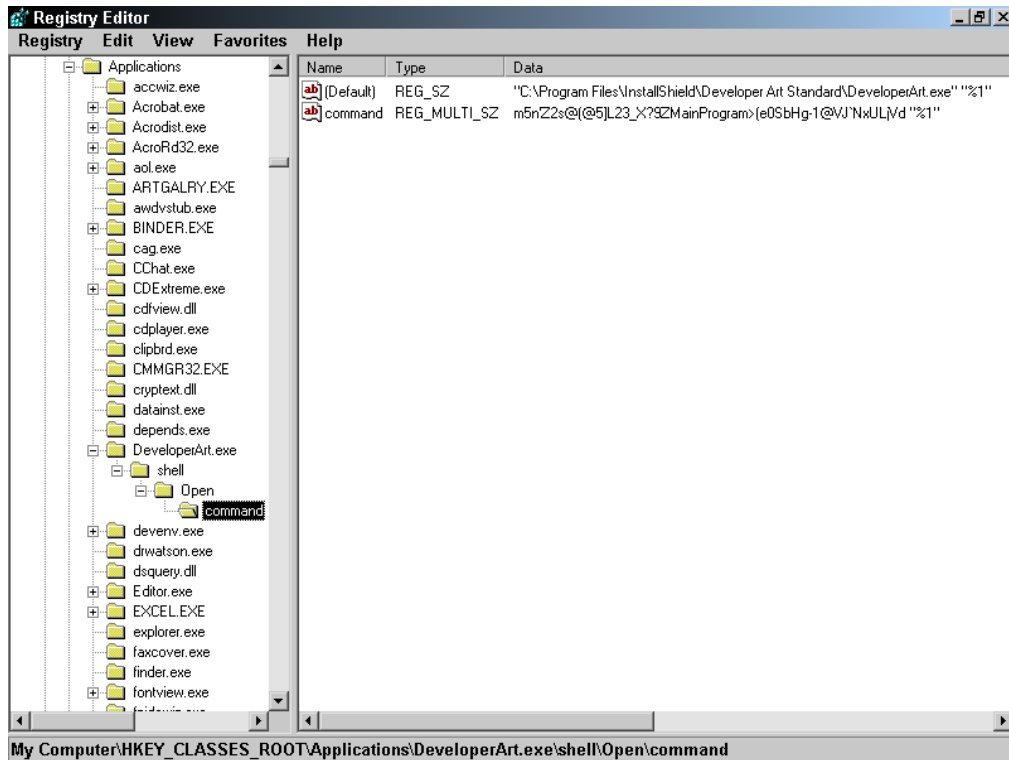


Figure 10-9: *The Applications key for the Developer Art file association.*

Under this key is written a key that has the name of the extension server and under this key is the same Shell\Open\command key structure that appears under the ProgID key discussed above. For the Developer Art file association, this entry in the registry is shown in Figure 10-9.

You have worked through the creation of a simple file association and have seen how it works. You now need to see what is necessary to extend this work so that if your file is attached to an email, a recipient can open the file with the correct application.

Adding a MIME Type to the File Association

The MIME (Multipurpose Internet Mail Extensions) standard was created to allow users to be able to manipulate files that are not natively supported by email applications or Web browsers. MIME works very much like file associations, as described above, but it uses something called a content type. A content type consists of a major type and a sub-type that are separated by a forward slash. When InstallShield Developer asks for a MIME type, it is asking for a content type. As an example the content type for a text file is `text/plain`. You have a fair amount of flexibility in the selection of a content type (MIME type) to use for your applications.

To create a MIME type in your project, perform the following four steps:

1. Go to where you defined the `idv` extension under the File Types node. Right-click on the `idv` file type and select New MIME Type. This creates a new MIME type with a default name.
2. Type the following for the MIME Type name:

```
application/x-devart
```

The major content type is `application`, which means that it is a non-standard file format. The content sub-type is a unique name that is specific to the Developer Art application.

3. After entering the new MIME type, click the `idv` extension to display its property grid. From the MIME Type property drop-down menu, select the `application/x-devart` MIME type.

- Click on the application/x-devart MIME type you just created to display the property grid. Because DeveloperArt.exe is not a COM server, you do not have to enter anything for the Class ID property. The value of the Class ID property is used to populate a row in the MIME table.

When you are finished adding the MIME type to your project, you will see what is shown in Figure 10-10.

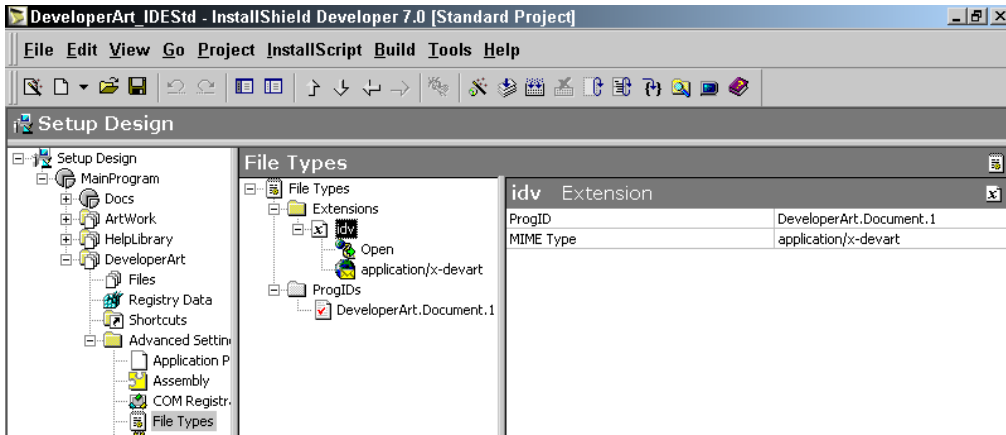


Figure 10-10: *The MIME type for the .idv extension in the Developer Art application.*

To test the inclusion of a MIME type in your application, build the project and install the Developer Art application. When you run the installation, two new entries are made in the registry in order to support the new MIME type. The first added entry is a new value that is written against the extension key under HKEY_CLASSES_ROOT (Figure 10-11).

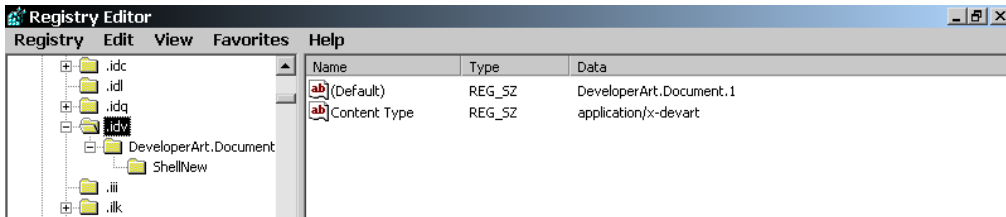


Figure 10-11: *The Content Type entry in the registry for the Developer Art MIME type.*

Note that there is now a value name called Content Type and it has a value data entry equal to the MIME type that you created in your project. The second entry that is made in the registry is the addition of your MIME type to the MIME database. This MIME database is located under the following key:

```
HKEY_CLASSES_ROOT\MIME\Database\

```

The entry under this key made by the installation of the Developer Art application is shown in Figure 10-12.

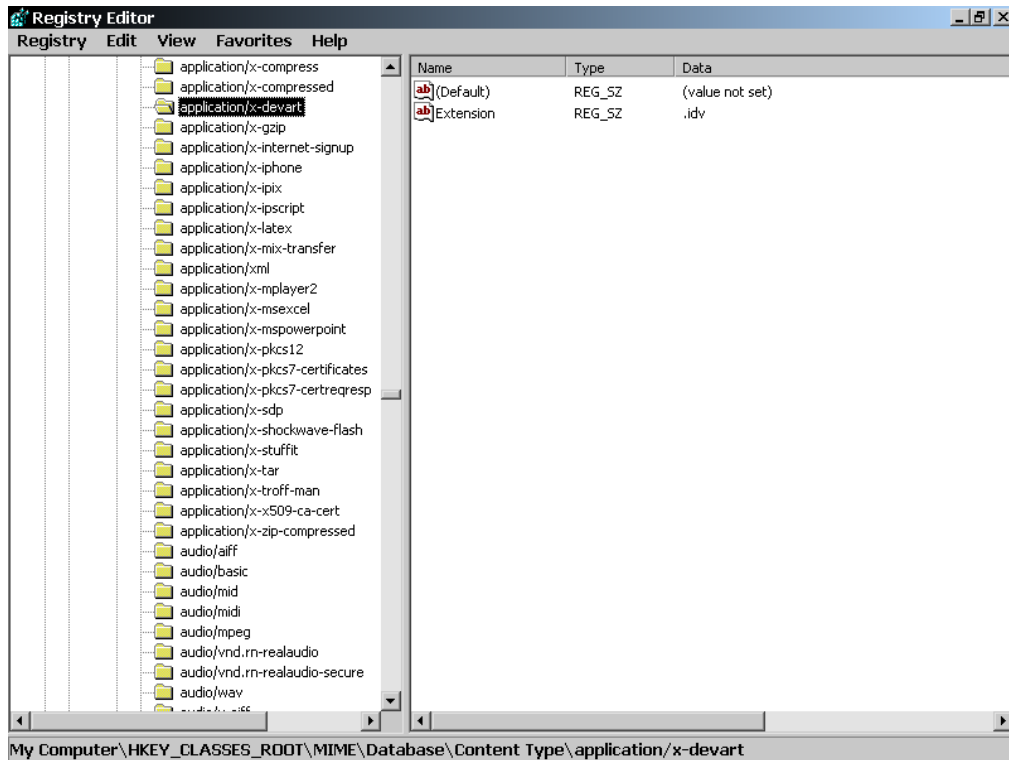


Figure 10-12: *The entry in the MIME database in the registry.*

The final test that you need to run is to attach your .idv file to an email and send it to yourself. You should see the attached file with the correct icon beside it. Double-click on the attached file and select the “Open it” option on the “Opening mail Attachment” dialog. The Developer Art application should open this attached file. If

you use Microsoft Outlook as your mail client, you should see something like what is shown in Figure 10-13.

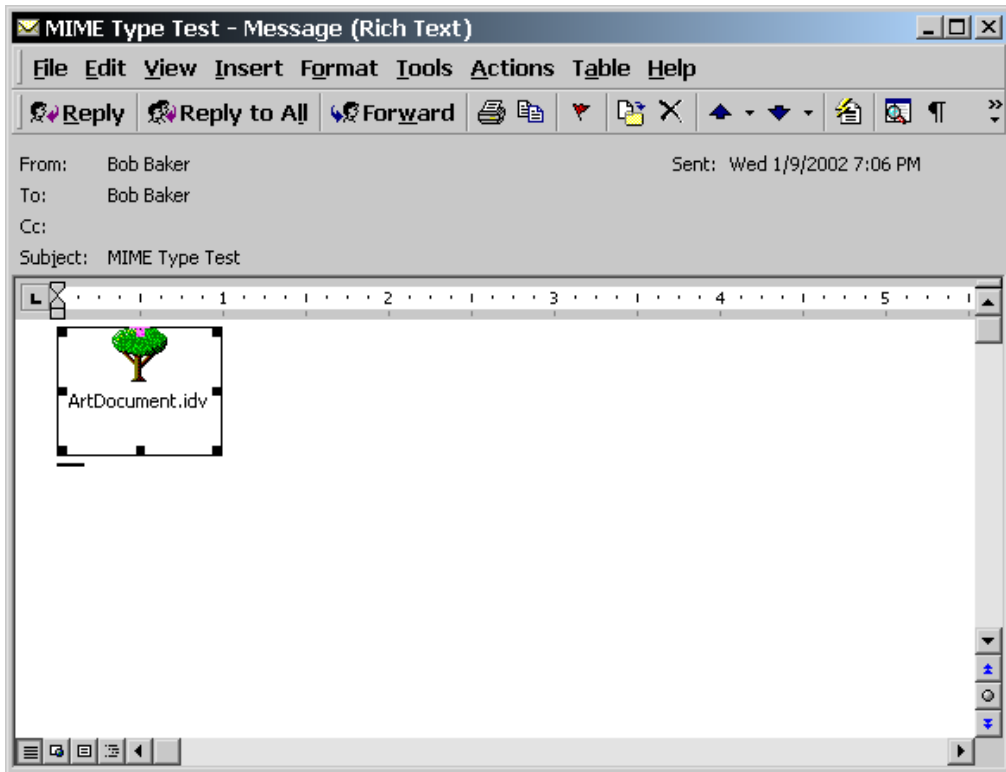


Figure 10-13: *The .idv file as an attachment in an email.*

Defining Registry Entries

A major part of an application installation involves making entries in the registry. This was demonstrated in the last section when you created a file association and a MIME type for the file type that is created by the Developer Art application. You are also creating registry entries when you install a COM server such as the ArtWork component. To make these particular registry entries, a number of special tables are used. None of these entries are defined in the Registry table.

The Registry table is used to make registry entries of a generic nature. Chapter 5 discussed one of the registry entries that is created by an entry in the Registry table. This particular registry entry is the key that is written under the following location:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\  
CurrentVersion\App Paths
```

Because of this key's importance, there is a special Application Paths icon under each component's Advanced Settings in either a Standard project or a Basic MSI project where this particular registry entry is defined. The information entered here is built into the Registry table.

This section discusses the definition of registry entries in the Registry table. Since the approach used is the same for both a Standard project and a Basic MSI project, you can use the Standard project that you used in the last section to define a file association for the Developer Art application. In this discussion, you will not use any of the built-in registry functions available in InstallScript because we want to stay away from the programmatic approach to working with the registry. The danger of using the registry functions is that, in a managed environment, these entries might not be created during the installation. Accordingly, it is better to define the required registry entries by creating rows in the Registry table.

To work with the Registry table in InstallShield Developer, it is important to understand the various columns that make up this table. You also need to know about the special functionality provided by the RemoveRegistry table. This is the subject of the next section.

The Registry and RemoveRegistry Tables

The Registry table is used to define keys and values that are created during an installation. You can also define special functionality that is applicable to registry keys during an uninstallation. The RemoveRegistry table is used to define registry keys and values that are to be removed during an installation. For both tables, the actions specified are associated with a particular component and these actions are executed only if the associated component is selected for installation or uninstallation. Making entries in the Registry table is fully supported in InstallShield Developer's IDE through context menus and drag and drop functionality. To add rows to the RemoveRegistry table, you need to use the Direct Editor, which is found under Advanced Views.

The Registry Table Schema

There are six columns in the Registry table. The format of these columns is described in Table 10-1.

Table 10-1: The Registry Table Schema

Column Name	Description
Registry	This column serves as the table's primary key. This is a string that adheres to the requirements of the Identifier data type. This entry needs to be unique for each row in the table.
Root	This column contains numerical values that identify the predefined root key for the key and/or value that is to be created. The valid values are described in Table 10-2.
Key	This column defines the key that is written to the registry. This is a string that adheres to the requirements of the RegPath data type. This string consists of the key hierarchy from just below the root key to the key that is to be created or under which values are to be written. For example, the entry in this column for the creation of the App Paths entry is as follows: SOFTWARE\Microsoft\Windows\CurrentVersion\ App Paths\DeveloperArt.exe
Name	In this column, you define the value name that is to be created under the registry key specified in the previous column. This column uses the Formatted data type. If this field is NULL, the value in the next column is written as the data for the default name. Using the string in this column, you can control what happens to a key during installation and uninstallation as long as the Value column is NULL. The special formatting characters are described in Table 10-3.

Table 10-1: The Registry Table Schema (Continued)

Column Name	Description
Value	The string that is placed in this column must also conform to the Formatted text requirements. This is the data that is associated with either the string placed in the Name column or the default value for a key. The special formatting strings used in this column are described in Table 10-4.
Component_	The value in this column is a foreign key into the first column of the Component table. This ties any registry changes defined in a row of the Registry table to whether the associated component is installed or uninstalled.

If you look at the Registry table in the Direct Editor you will see that there is a column that is not described in Table 10-1. The name of this column is ISAttributes and the purpose of this column is to manage how the registry entries are displayed in the IDE. This column is not included in the .msi file when you run a build.

An understanding of what goes into the Registry table is important so you will know what the context menu in the IDE is doing for you when you define registry entries to be created when a component is installed. Because of this we look at the possible special characters for the Root, Name, and Value columns of the Registry table. The Windows Installer performs certain actions based on what special characters are used in these columns.

In the Root column of the Registry table there are specific integer values that are meaningful. The possible values in this column are described in Table 10-2. Most of the values shown in Table 10-2 are used in other tables when it is necessary to identify the root registry key. Of particular interest is the -1 value, which has a special meaning only to installations run on Windows 2000 and Windows XP. In the IDE you will make use of this special new functionality by adding keys and values to the HKEY_USER_SELECTABLE root key. Of course there is no such key actually in the registry, only in the IDE.

Table 10-2: Valid Values for the Root Column

Value	Meaning
-1	If the installation is being performed for the current user, the key and/or values are created under HKEY_CURRENT_USER. If the Installation is being performed for all users of the machine, the key and/or values are created under HKEY_LOCAL_MACHINE.
0	The keys and/or values are created under HKEY_CLASSES_ROOT.
1	The keys and/or values are created under HKEY_CURRENT_USER.
2	The keys and/or values are created under HKEY_LOCAL_MACHINE.
3	The keys and/or values are created under HKEY_USERS.

You can control certain aspects of how a branch in the registry is treated during installation and uninstallation by applying certain formatting characters in the Name field for a leaf key of the branch. The formatting characters are shown in Table 10-3.

Table 10-3: Formatting Characters for the Name Field

Character	Meaning
+	This symbol, used as the value name, creates a registry key when the associated component is installed. This key will not be uninstalled as long as there are no other values associated with this key.

Table 10-3: Formatting Characters for the Name Field (Continued)

Character	Meaning
-	This symbol used as the value name removes a key and all its sub-keys and values when the associated component is uninstalled. It does not matter if the key already existed before the installation was run.
*	This symbol used as the value name creates a key when the associated component is installed and removes the key when the associated component is uninstalled.

In the Value column of the Registry table there are a number of special strings used depending on the storage format of the data being created. The special strings used in this column of the Registry table are described in Table 10-4.

Table 10-4: Formatting Characters for the Value Field

Storage Format	Special String Description
REG_BINARY	To create a registry value with this storage format the value in the Value column needs to be prefixed with #x.
REG_EXPAND_SZ	To create a registry value with this storage format the value in the Value column needs to be prefixed with #%.
REG_DWORD	To create a registry value with this storage format the value in the Value column needs to be prefixed with #.
REG_MULTI_SZ	To create a NULL-delimited list of strings, place a tilde inside square brackets between each string such as with "a [~] b [~] c.". This will replace any values already in the registry for the key identified in the Key column.

The NULL terminator ([~]) used to create the REG_MULTI_SZ storage type is also used to identify whether an entry in the Value column of the Registry table is to be added to what is already in the registry. If the NULL terminator is prefixed to the string in the Value column then the string is appended to what is already in the registry. If the NULL terminator is appended to the string in the Value column then the string is added at the beginning of what is already in the registry. In either case if the string being added to the registry is already present then the string in the registry will be removed. If the NULL terminator is placed at both the beginning and the end of the string in the Value column then any existing value in the registry will be replaced.

If there are no special strings used in defining the entry in the Value column of the Registry table, the Windows Installer interprets the value to be written to the registry as the REG_SZ storage format.

The RemoveRegistry Table Schema

There are five columns in the RemoveRegistry table. There is no Value column in this table because the sole purpose of this table is to remove keys, sub-keys, and values during the installation of an associated component. The schema of the RemoveRegistry table is described in Table 10-5.

To generate entries in the RemoveRegistry table, use the Direct Editor view under Advanced Views in the InstallShield Developer IDE. Using the Direct Editor is similar to using Orca. Orca is the database-editing utility that comes with the Windows Installer SDK.

Table 10-5: The RemoveRegistry Table Schema

Column Name	Description
RemoveRegistry	This column serves as the primary key for this table. This is a string that adheres to the requirements of the Identifier data type. This entry needs to be unique for each row in the table.

Table 10-5: The RemoveRegistry Table Schema

Column Name	Description
Root	This column contains numerical values that identify the predefined root key for the key and/or value that is to be removed. The permissible values are the same as described in Table 10-2 for the Registry table.
Key	The value specified here defines the registry key under which the value defined in the next column is to be removed during an installation. If the next column contains a minus sign (-), this key and all its values and sub-keys are removed during an installation. The value in this column needs to conform to the RegPath data type requirements.
Name	This column defines the value name that is to be removed from the registry key specified in the previous column. This column uses the Formatted data type. If you place a minus sign (-) in this column instead of an existing value name, the key specified in the previous column, with all its sub-keys and values, is removed during an installation of the associated component.
Component_	The value in this column is a foreign key into the first column of the Component table. This ties any registry changes defined in a row of the RemoveRegistry table to the installation of the associated component.

Note that the use of the minus sign is required in Name column so that a registry key with all its values and sub-keys will actually get removed during an installation. Without the minus sign only the named value will get removed.

Working with the Registry Table

This section allows you to experiment with the functionality in the IDE that is available for defining rows in the Registry table. When you define registry entries that need to be made during an installation, you define these registry entries relative to a particular component. There are two separate locations in the IDE where you can define registry entries. There is a global view under Step 3 where you can see all the registry entries that are being made by all components or you can filter the view so only the registry entries being made by a single component are visible. You can also go to the Setup Design view under Advanced Views and see the registry entries that have been defined for an individual component.

Both the Setup Design under Advanced Views and the Registry view under Step 3 have the same functionality. The top two panels show the registry on the build machine and the bottom two panels show the registry entries that are defined in the project. For this chapter, use the global location for defining and viewing registry entries. However, under each component in the Setup Design view, there is an icon called Registry Data. You can do the same operations here as you can in the global location under Step 3.

Click on the Registry view under step 3 in your Standard project for the Developer Art application to the global Registry view (Figure 10-14). Figure 10-14 shows the view when the View All Entries (read-only) option is selected. All the registry entries that are being created by entries in the Registry table are shown. At this time, the only registry entry that should be seen is the values written under the App Paths registry key. This registry value was defined in Chapter 5. The Developer Art application does not require any other registry entries to be created from the Registry table. To see how to use the functionality in the IDE, you need to create some arbitrary keys and values.

There are a number of ways to define registry entries in your project. One method is to drag and drop registry keys from the registry on the build machine into the project. To do this, navigate in the upper-left panel to the key that you want to incorporate into your project. Drag that key from the upper-left panel that shows the registry keys on the build machine to the lower-left panel that displays the registry entries defined in the project. You can also drag value names and their data from the upper-right panel down to a particular key in the lower-left panel. Additionally, you can drag value names and their data down to the lower-right panel and these will be associated with

the key that is highlighted in the lower-left panel. Drag and drop works only when a particular component is selected. When the View All Entries option is selected, the view is in a read-only mode and you cannot create any registry entries.

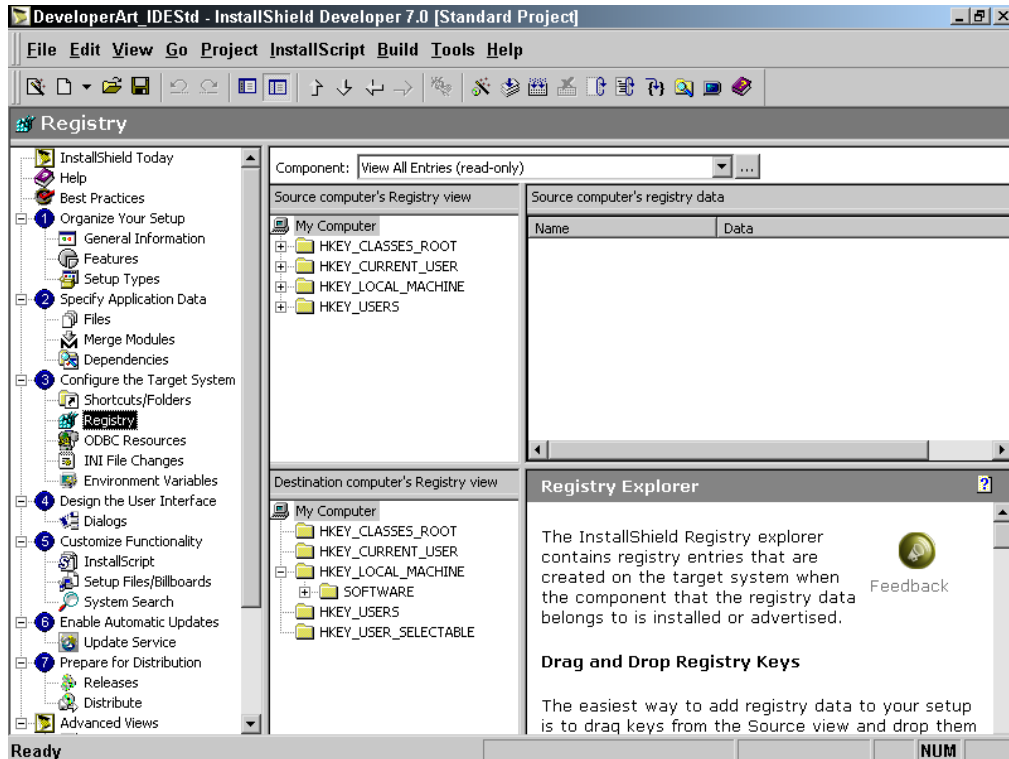


Figure 10-14: *The global registry view in the Developer Art project.*

You can add registry entries to your project by importing a .reg file. To import a .reg file follow the two steps listed below:

1. Right-click on any registry key and select Import REG File. This launches the Import REG File Wizard.
2. Follow the wizard panel instructions. You will name the .reg file to import and indicate how conflicts should be handled if there are already values defined in the project that are also defined in the .reg file.

Except for the drag and drop functionality, all the means to define registry entries in your project come from the context menu when you right click on a key in the lower-left panel or when you right click anywhere in the lower-right panel. The context menu when you right click on a key in the lower-left panel is shown in Figure 10-15.

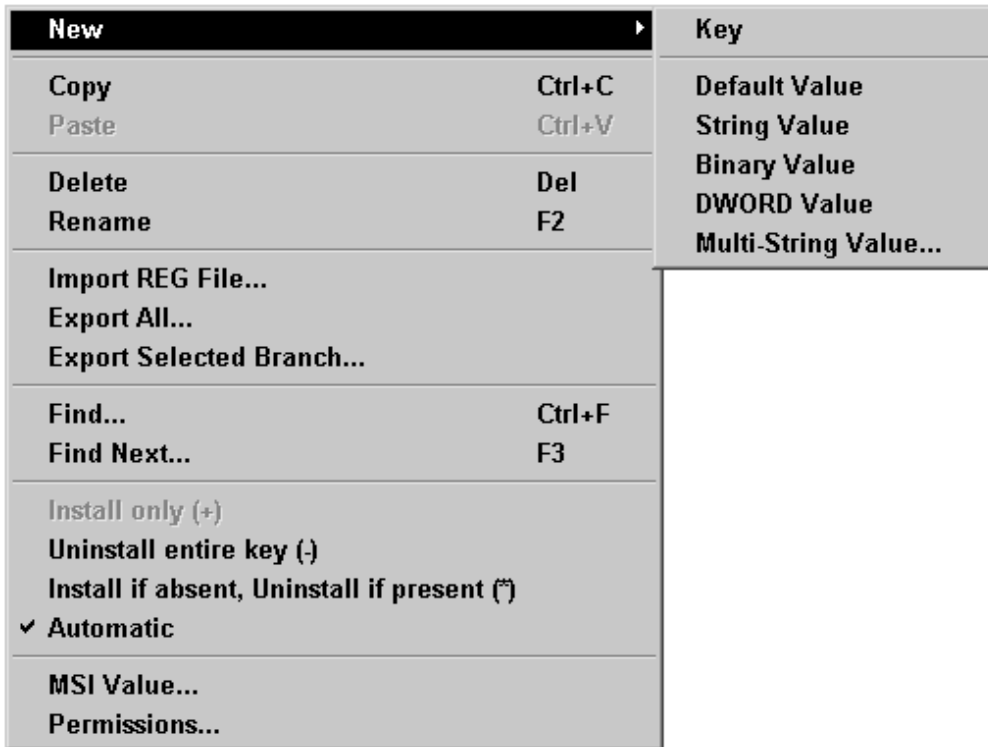


Figure 10-15: *The context menu for creating registry entries in the Destination computer's Registry view panel.*

The various options on the context menu are discussed in the following list. The operations on this menu that are key specific are valid only in the lower-left panel of the Registry view. Those operations that can apply to both keys and values can also be accessed from the context menu in the lower-right panel

New: This option has a sub-menu that allows you to create a new key under the highlighted key highlighted. You can also create one of five types of data for a key. Selecting the Multi-String Value option launches a dialog box that permits you to create a REG_MULTI_SZ string. This dialog will give you the

opportunity to place the NULL-terminator in front or at the end of the value you are defining.

Copy & Paste: You can copy and paste keys with their values and data from one place to another in the lower-left panel. You cannot copy and paste individual values and their data.

Delete & Rename: With these two options, you can delete entries that you have defined and you can rename values and keys. The renaming functionality is not case sensitive, so you change the case of a letter to rename an item.

Import REG File: This option launches the Import REG File Wizard, which allows you to choose a .reg file to import into your project.

Export All & Export Selected Branch: After performing all the work to define the registry entries you want generated when a component is installed, you can capture that information for later use by exporting it to a REG file. You can elect to export everything or just part of the information by selecting only a branch of the registry hierarchy.

Find... & Find Next...: These options provide a search mechanism for finding a keys, values, or data in the registry entries that have been created in your project.

Install only (+): For a leaf key, when the key does not have any value assigned, you can have the key created during an installation but can prevent the key and all its ancestors from being removed when the associated component is uninstalled. This option cannot be used on non-leaf keys or where the leaf key has any other value assigned to it. Another means to create registry entries that do not get uninstalled is to create a component that will create these entries and mark the component as permanent so that it will never get uninstalled.

Uninstall entire key (-): This is the opposite of the previous option and it can be used on key in a branch. Here the key and its values are not created during an installation, but if the designated key already exists on the target system, it is uninstalled when the associated component is uninstalled.

Install if absent, Uninstall if present (*): This is almost like standard functionality where the registry entries are made when the associated component is installed and removed when the associated component is uninstalled. The

difference here is that the registry entries that are uninstalled do not need to have been created when the component was installed. If the subject registry entries already exist on the target system prior to the component's installation, they are removed when the associated component is uninstalled.

Automatic: This is the standard operation where only the registry entries that are associated with a component are removed when the component is uninstalled. If the registry entries already exist prior to an installation, the registry entries will not be removed when the associated component is uninstalled. This is the safe approach where a component does not remove anything that it does not place on the system. However, if a registry entry is overwritten during an installation this entry will be removed when the associated component is uninstalled. The old value of the registry is not replaced.

MSI Value: The first column of the Registry table is the primary key for the table (Table 10-1). Under normal operation, the InstallShield Developer build process generates a unique value for this column. To access this table during an installation using a custom action, you need to know the primary key for the row that you want to access. Using this option you can replace the unique name generated by InstallShield Developer with a unique name of your own. This allows you to code your custom action in advance by using the primary key.

Permissions: This option launches the Permissions dialog that you can use to set permissions for a registry entry. The settings that you set with this dialog are used to populate the LockPermissions table. The subject of setting permissions is outside the scope of this book.

Figure 10-16 shows the context menu that is launched when you right click in the lower-right panel of the Registry view. This menu is used to work with the value names and value data under the key that is selected in the lower-left panel.

The top five options on the context menu have the same functionality as the similarly named options on the New sub-menu shown in Figure 10-15. When you select to create a particular type of value in the lower-right panel, a default name is provided for this value, which you can rename. To rename a value you can select the Rename option on the context menu, you can slowly click twice on the value name, or you can hit the F2 function key.



Figure 10-16: *The value names and value data context menu.*

Once you give a name to the value, you then need to provide data for this value. Do this by right clicking on the value name and selecting Modify. This launches a dialog where you enter the data for the value. You can also launch this dialog by double clicking on the value name. When you create a DWORD value you get default data equal to 0. When you create either a String value or a Binary value the data value is initially set to NULL.

When you choose to create a Default value, the data field is automatically set to the string "(value not set)". However, when the Windows Installer creates the registry key the Default value will be NULL, not "(value not set)". This is a bug so it is better not to create a Default value in your project unless you intend to set the data for this value to something specific.

At the bottom of the context menu shown in Figure 10-16 are two options that are used to either set a registry entry as the key path for a component or to remove a registry entry from being the key path for a component. As discussed in Chapter 3, it is possible to identify a registry entry as the key path for a component.

Adding Registry Entries to the Developer Art Project

One of the standard registry entries that are made for an application consists of some basic information including the company name, product name, and product version. Usually values that are important for the product that is installed are written as values for the version key. In the following example, you will do something similar for the Developer Art installation. When you create these registry entries, you will use the formatted data type to capture the name and values that you want to write to the registry. The formatted data type means that you can enclose the name of a property in square brackets and the string will be replaced at installation time with the value of the property.

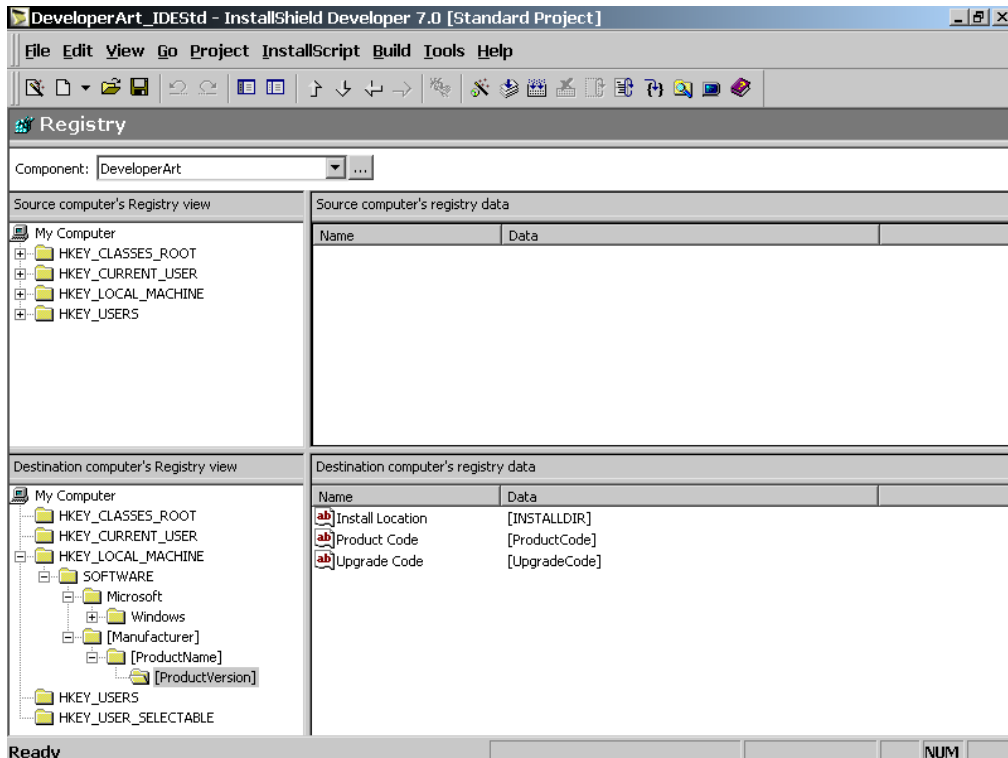


Figure 10-17: Registry entries for the Developer Art application.

This example writes all of the values to HKEY_LOCAL_MACHINE. For the example, you will associate these registry entries with the DeveloperArt component. This is shown in Figure 10-17.

As shown in Figure 10-17, three different private properties are used to create the three keys: Manufacturer, ProductName, and ProductVersion. These properties are enclosed in square brackets and thus the actual value of the property is used to create the name of the registry key. Table 10-1 explained that the Key column of the Registry table has a data type of RegPath. This data type accepts formatted strings for creating parts of the total registry branch.

In Figure 10-17 shows three value names with data under the ProductVersion key. The value names are strings for the installation location and values of the ProductCode and UpgradeCode properties. For the Install Location value name, the data is the value of the INSTALLDIR public property.

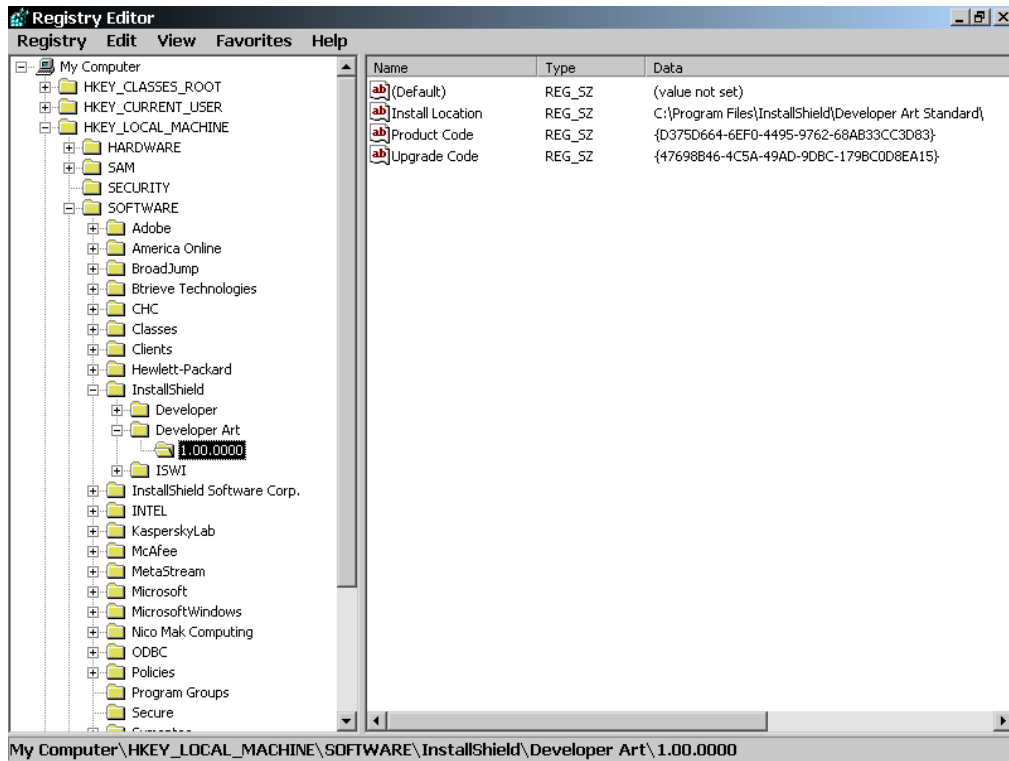


Figure 10-18: Registry entries created by the project input shown in Figure 10-17.

After you make these entries for the DeveloperArt component, you need to build the project and install the application to verify that the entries in the registry have been made properly. The registry entries that are created are shown in Figure 10-18.

Before moving on to a look at the RemoveRegistry table, you can experiment a little with some of the special functionality provided by the Windows Installer. First, see how you can prevent registry keys from being uninstalled when the Developer Art application is uninstalled. To do this:

1. Add an arbitrary key under the product version key.
2. Right-click on this key and select “Install only (+)” (Figure 10-19). Figure 10-19 shows a registry key named Key under the product version key. This key has a plus sign (+) on it, indicating that it will be installed, but not uninstalled. To test this, build the project, run the installation, and then uninstall the application. When you look in the registry, you see that all the keys are still there but the value names and value data associated with the product version key have been removed. This demonstrates that you can prevent only keys from being uninstalled, but not the associated values and data.

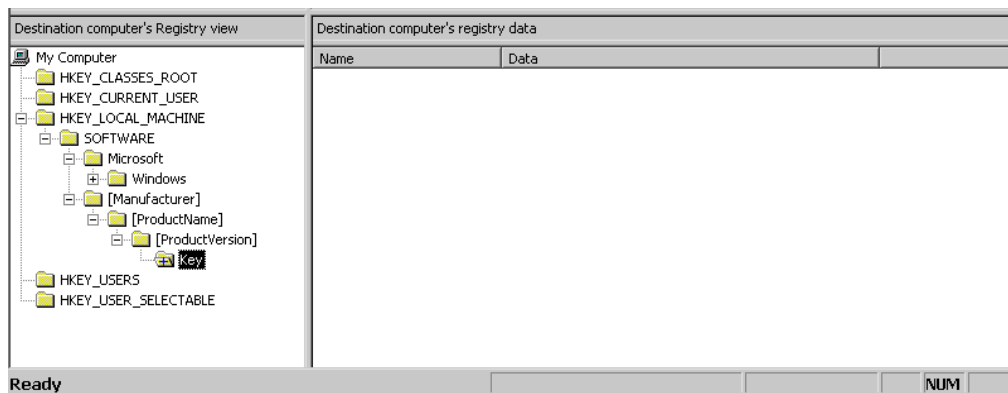


Figure 10-19: Preventing keys from being uninstalled.

Now that there are some registry entries on the system, you can see what the “Uninstall entire key (-)” option does. Right-click on the product name key and select the “Uninstall entire key (-)” option. For this example, you will remove an entire branch. To do this, go to the top of the branch you want to remove, right-click, and

select “Uninstall entire key (-)”. You do not have to reverse the “Install only (+)” on the leaf node because the “Uninstall entire key (-)” option takes precedence. When you select this option on the product name key, you will see something like what is shown in Figure 10-20.

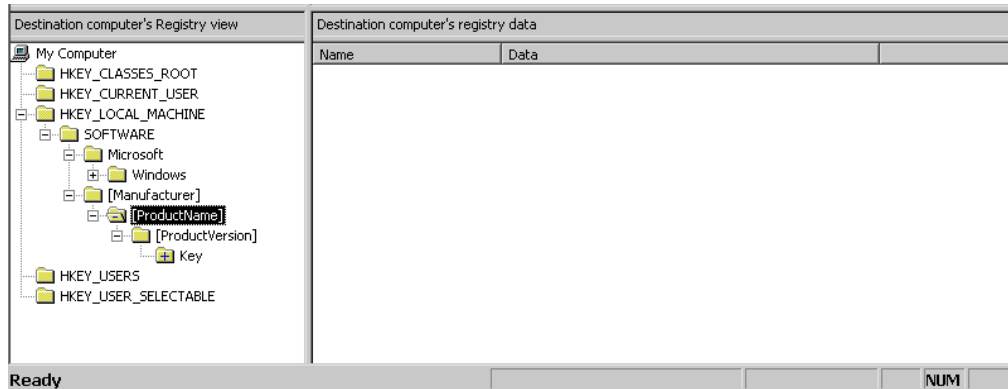


Figure 10-20: Forcing the removal of a branch in the registry.

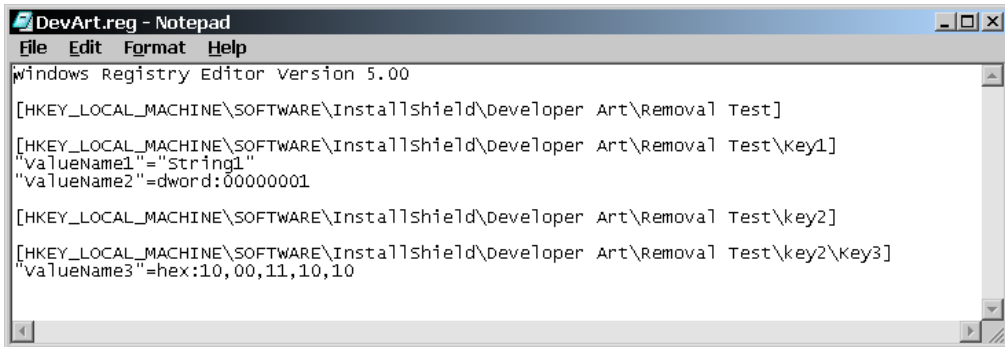
To test this, build the project, install the application, and uninstall the application. After uninstalling the application, go into the registry to see that this entire branch has been removed.

Removing Registry Entries During an Installation

The previous section dealt with the Registry table. The IDE has a number of options that make it easy to properly populate this table. The IDE does not provide context menu functionality for the RemoveRegistry table. To populate the RemoveRegistry table, you need to use the Direct Editor view under Advanced Views. The Direct Editor contains a list of all the tables represented in the project file. Scroll to the table you want to edit and make entries according to the Windows Installer help for the table.

In order to run an example of the RemoveRegistry table, you need to create some keys and values in the registry before you run the installation. The best way to do this is to use a REG file to add these keys. The CD-ROM contains a REG file named

DEVART.REG under the Chapter 11\Sources\Developer Art folder. The contents of this file are shown in Figure 10-21.



```

windows Registry Editor Version 5.00

[HKEY_LOCAL_MACHINE\SOFTWARE\InstallShield\Developer Art\Removal Test]

[HKEY_LOCAL_MACHINE\SOFTWARE\InstallShield\Developer Art\Removal Test\Key1]
"ValueName1"="String1"
"ValueName2"=dword:00000001

[HKEY_LOCAL_MACHINE\SOFTWARE\InstallShield\Developer Art\Removal Test\key2]

[HKEY_LOCAL_MACHINE\SOFTWARE\InstallShield\Developer Art\Removal Test\key2\key3]
"ValueName3"=hex:10,00,11,10,10

```

Figure 10-21: The .reg file for adding entries to the registry.

You need to add the necessary information to the RemoveRegistry table so the registry keys and values defined in the .reg file (Figure 10-21) are removed during an installation. To do this, remove the branch that consists of all keys and values starting with the key named "Removal Test".

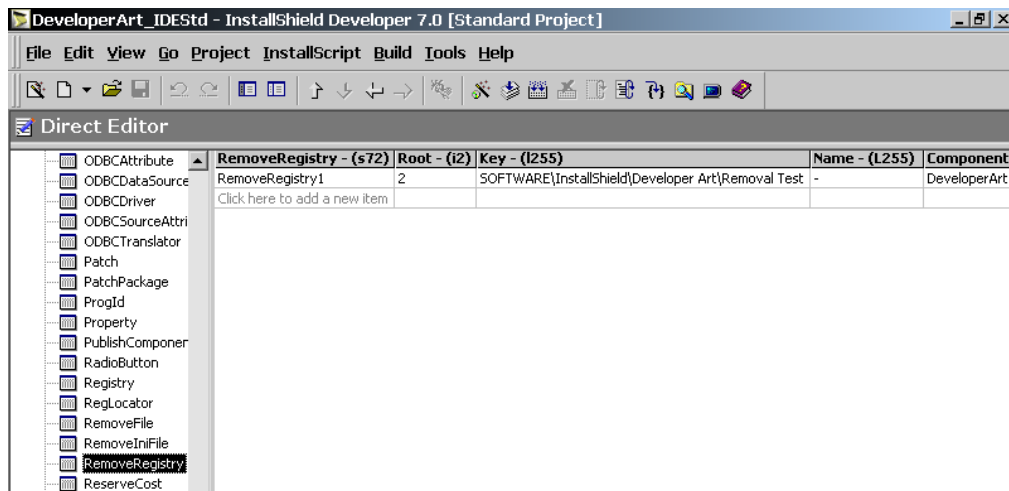


Figure 10-22: The Direct Editor view showing the entries in the RemoveRegistry table.

Refer to Figure 10-21 and Table 10-2 to define one row in the RemoveRegistry table using the Direct Editor view. The entries that you need to make in the RemoveRegistry table using the Direct Editor view are shown in figure 10-22.

Click on the first row in the RemoveRegistry table. A default primary key is provided for the first column. Leave the default value in this column. Since you are concerned with removing keys and values under the HKEY_LOCAL_MACHINE root, type 2 in the second column (Root).

Define the location of the key that you want removed during an installation in the Key column. This value begins with the SOFTWARE key and ends with the name of the key that is to be removed. Note that a back slash is not required at the beginning nor at the end of the value placed in the Key column.

In the Name column, place a minus sign to signify that you want all sub-keys and values removed. If you want to remove only values, you would place the value name in this column. In this case, a separate row is required for each value that you want removed.

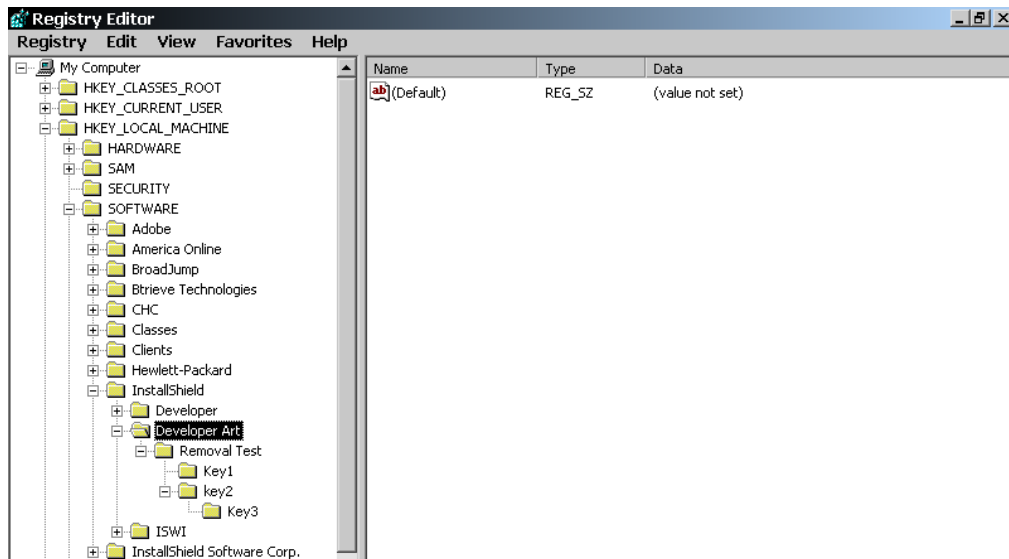


Figure 10-23: *After merging DevArt.reg but before installing Developer Art.*

In the last column, select the component that needs to be installed before the desired removal takes place. The Direct Editor provides a drop-down menu that lists the project's components.

Before testing this functionality, you first need to merge the DEVART.REG file with the registry. To do this, right click on the .reg file and select Merge. After this .reg file has been merged, but before the Developer Art application is installed, the registry appears similar to what is shown in Figure 10-23.

After making the entries in the RemoveRegistry table and building the project, install the Developer Art application. After the installation is complete, look at the registry to see that the registry entries shown in Figure 10-23 have been removed.

Handling Environment Variables

An installation often has to perform tasks related to environment variables, including setting environment variables during the installation. Using custom actions, you can also retrieve the value of an environment variable, and set environment variables that exist only while the installation is running.

When defining environment variables that are to be created during an installation, the definition needs to be associated with a component. The defined environment variable is created only if the associated component is installed. When you define an environment variable, you are making entries in the Environment table. Using this table you can create new environment variables, append new values to existing environment variable values, and remove environment variables. The InstallShield Developer IDE contains an Environment Variables view, which allows you to author rows in the Environment table without using the Direct Editor view. The Environment Variables view is under Step 3 and is the same for both Standard and Basic MSI projects.

The first thing that we want to do is take a brief look at how environment variables are handled, with particular emphasis on Windows 2000.

Environment Variable Overview

There are two types of environment variables on Windows NT/2000/XP, user environment variables and system environment variables. User specific environment variables can be different for each user. System variables are the same regardless of what user is signed on to the machine. Only users that are members of the Administrator's group can add new system environment variables or change the values for existing variables.

On Windows 2000, system environment variables are stored in the registry at the following location:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\  
Session Manager\Environment
```

User environment variables are stored in the registry at the following location:

```
HKEY_CURRENT_USER\Environment
```

When an application is launched on Windows 2000, a new process is created and in this new process a block of memory is allocated where the values of the currently defined environment variables are stored. New environment variables that get set while this application is running are not added to this environment block and as such the application does not know anything about them until it is shutdown and re-launched.

The loading of environment variables into an applications memory space is performed in the following order:

- System environment variables are set first.
- User environment variables are set next and if there is any conflict with the system variables the user variables will override the value of the system environment variable.
- Environment variables that are defined in Autoexec.bat are set last. These variables do not override any of the system or user environment variables.

There is one exception to the above rules and that is the `PATH` environment variable. The values for this environment variable are cumulative the paths defined in all three locations are appended to each other.

When working with InstallShield Developer, it is important to remember that each process has its own set of environment variables stored in the environment block. You saw in Chapter 4 that there are a number of processes running during any installation. If one of these processes sets an environment variable the other processes will not see it and not have access to it.

Now it is time to get into some detail about working with environment variables in InstallShield Developer. To help you better understand the Environment Variables view, the next section examines the Environment table schema.

The Environment Table Schema

The Environment table contains four columns (Table 10-6). To view the Environment table, go to the Direct Editor. Note the use of prefixes in the Name column to control how environment variables are created or removed.

Table 10-6: The Environment Table Schema

Column Name	Description
Environment	This is the primary key for the table and it needs to conform to the Identifier data type. The value in this column must be unique for each row in the table.
Name	This column contains the name of the environment variable that is to be manipulated during the installation. How the environment variable is treated during an installation or an uninstallation depends on the prefix that is used on this name. This name is not case sensitive. When the only difference between two names is the case of the letters used, they are considered the same environment variable. Table 10-7 describes the valid prefixes and the functions they serve.

Table 10-6: The Environment Table Schema (Continued)

Column Name	Description
Value	<p>This column contains the value that is to be assigned to environment variable specified in the Name column. The string that is placed in this column needs to conform to the Formatted data type specification.</p> <p>When a value is to be appended to an existing value, use the tilde inside square brackets [~], which, as discussed earlier, represents the NULL character. You also use the NULL character to prefix a value to an existing value. To append a value to an existing value, enter the following in the Value column, [~];NewValue. Note that the semi-colon delimiter is included as part of this string. To prefix a value to an existing value, reverse the position of the NULL character and the delimiter, as follows: NewValue;[~].</p>
Component_	<p>The value in this column is a foreign key into the first column of the Component table. This defines the component that needs to be installed before the environment variable defined in this row is created, modified, or removed.</p>

The string in the Name column must contain one or more prefixes. The environment variables are written or removed depending on the characters that prefix the name of the environment variable being manipulated. The valid prefixes and their meaning are discussed in Table 10-7.

Table 10-7: Valid Prefix Characters for the name Column

Prefix	Meaning
!-	This prefix is used to remove an environment variable during an installation or an uninstallation.

Table 10-7: Valid Prefix Characters for the name Column (Continued)

Prefix	Meaning
-	Remove the environment variable when the component is uninstalled. This removes only an environment variable that is created during the installation and does not remove one that already exists at installation. If you prefix or append a value to an existing value, then only the value added during the installation is removed when the associated component is uninstalled.
!	Remove the environment variable during an installation. The Windows Installer removes an environment variable during an installation only if the actual name and value of the existing environment variable matches the entries in the Name and Value fields of the Environment table. To remove an environment variable during an installation, regardless of its value, you would use this prefix with the name of the environment variable and leave the Value column empty.
*	This prefix is used with Windows NT and Windows 2000 to specify that the environment variable being set is a System environment variable and not a User environment variable. If no asterisk is present, the Windows Installer writes the variable to the user's environment. On Windows 95, 98, and Me, the asterisk is ignored and the environment variable is written to AUTOEXEC.BAT. On Windows 95, 98, and Me, the environment variable does not become available until the next system start.
+-	This prefix creates the environment variable during an installation if it does not already exist and removes the environment variable created during the installation when the associated component is uninstalled.

Table 10-7: Valid Prefix Characters for the name Column (Continued)

Prefix	Meaning
=-	The environment variable is set on installation and removed on uninstallation. If the environment variable exists at the time of the installation and you prefix or append a new value to the value that already exists, then only the new values are removed when the associated component is uninstalled. If you replace the existing value with the new value, then, when the associated component is uninstalled, the existing environment variable is totally removed. This prefix is InstallShield Developer's default implementation.
=	Create the environment variable if it does not exist. Set the value of the environment variable regardless of whether the environment variable already exists or has to be created during the installation. If the environment variable already exists you can replace it with a different value or you can prefix or append a value to the current value.
+	Create the environment variable if it does not exist and then set its value during an installation. If the environment variable already exists at installation, no changes are made to the value and you cannot prefix or append a value to the existing value.

The WriteEnvironmentStrings and the RemoveEnvironmentStrings standard actions process the rows in the Environment table. If these actions are not present in the InstallExecuteSequence table, then any environment variables defined in the Environment table will not be written, modified, or removed from the target system. Even after these actions run, the environment variables are not available to the installation process. However, a new process will have access to these new environment variables.

The next section looks at how the InstallShield Developer IDE manipulates the entries in the Environment table to implement the desired functionality.

Working in the Environment Variables View

Since both a Standard project and a Basic MSI project present the same IDE functionality, we will use the Standard project that you have been using in this chapter to discuss the Environment Variables view. To get started do the following:

1. Click on the Environment Variables view under Step 3 to see the sub-view tree that has no environment variables defined.
2. Right-click on the root icon of this tree and select Add Environment Variable to create an environment variable.
3. Name the variable EnvVar.
4. Click on this environment variable to display its property grid (Figure 10-24).

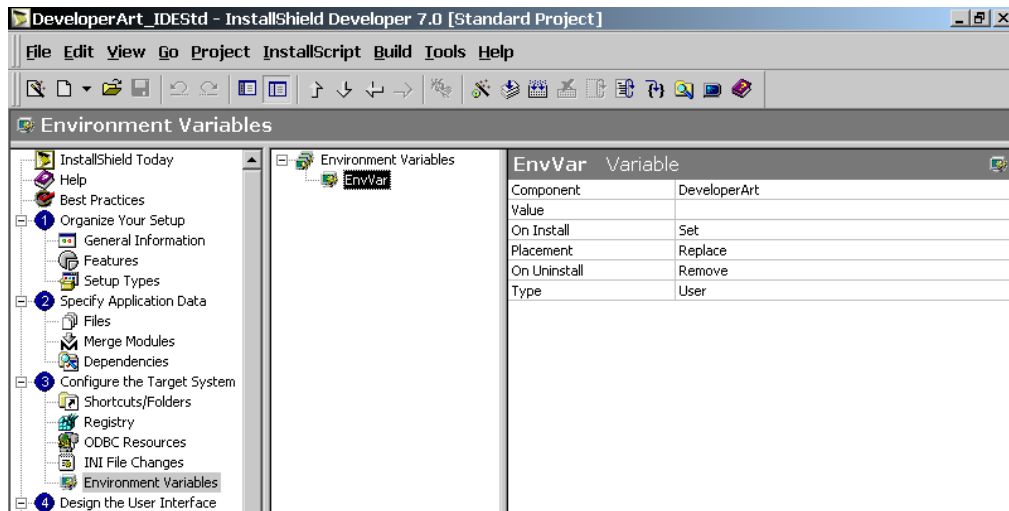


Figure 10-24: *The property panel for an environment variable.*

Figure 10-24 shows the six properties that determine the environment variable's functionality. By default, a new environment variable has the functionality that is defined by the `=-` prefix. This prefix creates and sets the environment variable during an installation and removes it when the associated component is uninstalled. The environment variable properties are discussed in the following list.

Component: Contains the name of the component that is associated with the environment variable that you want to manipulate during the application's installation and uninstallation. Click in the property field to display a drop-down menu that lists all of the project's components.

Value: Contains value for the environment variable that is to be set. If you are appending this value or prefixing it, you do not have to enter the delimiter and the NULL character. These are added as needed, based on the choices you make for the other properties.

On Install: This property specifies how you want the environment variable manipulated during an installation. The drop-down menu provides three choices for the property's value.

- **Set:** Places the equal sign (`=`) as part of the prefix in the name column.
- **Create:** Places the plus sign (`+`) as part of the prefix in the Name column.
- **Remove:** Places the exclamation mark (`!`) as part of the prefix to the value in the Name column.

Placement: Specifies whether to completely replace an existing value or to prefix or append the string in the Value field to an existing value. The drop-down menu provides the Replace, Append, and Prefix options for this property. If you select either Append or Prefix, and the environment variable does not already exist, the string in the Value field is made the value of the environment variable.

- **Replace:** This will cause any value already associated with the environment variable to be completely replaced.

- **Append:** This will cause the value to be appended to any existing value for the environment variable. If the environment variable does not already exist then this will set the value of the environment variable.
- **Prefix:** This will cause the value to be inserted at the beginning of any existing value for the environment variable. If the environment variable does not already exist then this will set the value of the environment variable.

On Uninstall: Specifies what to do with the environment variable when the associated component is uninstalled. The drop-down menu provides two options as follows:

- **Remove:** The minus sign (-) becomes part of the prefix to the value in the Name field.
- **Leave:** The prefix consists of only one character and that is the character that specifies what is to happen during an installation.

Type: Specifies for Windows NT/2000/XP where the environment variable is to be created. There are two possibilities as shown below:

- **User:** The user environment space makes an environment available to only the user that has performed the installation.
- **System:** The System environment space is available to all users. The end user must have administrative privileges in order to run an installation that creates or removes environment variables in the system environment space.

As discussed in Table 10-6, the Value column of the Environment table takes a Formatted text string as a value. This means that you can use property names inside square brackets to provide values for an environment variable. A simple example is to add the install location of the Developer Art application to the PATH environment variable.

Note that you need to use either the Append or the Prefix options for the Placement property when working with the PATH environment variable. If you were to completely replace this value it could damage your system.

5. Rename the EnvVar environment variable shown in Figure 10-24 to PATH. The component associated with the creation of this environment variable can remain the DeveloperArt component.
6. In the Value property, enter the INSTALLDIR property inside square brackets.
7. In the Placement property, select the Append option and leave all the other properties with their default values.

The final configuration for creating this environment variable is shown in Figure 10-25.

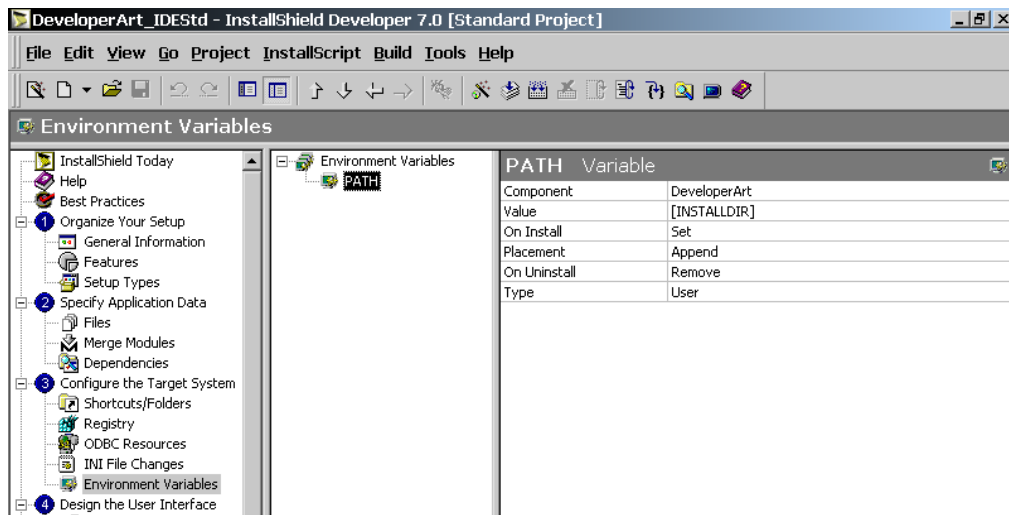


Figure 10-25: Adding a location to the PATH environment variable.

Note that you do not have to supply a delimiter or the NULL character in order for this Append operation to work correctly. This is handled by the Environment Variables view in the InstallShield Developer IDE. After you build the project and

install the application, you can go into the system dialog that displays the environment variables and see that the user PATH environment variable has the install location of the Developer Art appended to it. On a Windows 9x machine, you have to reboot the machine to see the change in the value of the PATH environment variable. This is because on Windows 9.x it is the Autoexec.bat file that is used to set environment variables. These variables get loaded into memory when the system boots and all applications have access to this memory.

When you uninstall the application, you see that only the value of the INSTALLDIR property has been removed from the value of the PATH environment variable. Experimenting with this functionality can provide a better understanding of the Windows Installer's capability for manipulating environment variables.

The Windows Installer help contains the following warning:

"Each row can contain only one value. For example, the entry *Value;Value;[~]* is more than one value and should not be used because it causes unpredictable results. The entry *Value;[~]* is just one value."

You can handle this situation by creating more than one entry for the same environment variable in the sub-view under the Environment Variables view. The creation of an environment variable name here translates into one row in the Environment table. Enter separate values for each of the instances of the same environment variable and make sure that to select either Append or Prefix for the Placement property. When the installation is run, semi-colons delimit all the individual values, and they form the value string for the environment variable.

On Windows NT, Windows 2000, and Windows XP, the end user usually has the opportunity to select to run the installation for all users of the machine or just for themselves. The next section discusses how this affects setting environment variables.

Per-Machine vs. Per-User Installations and Environment Variables

Environment variables that are generated during an installation need to be written to the System environment space so they are available to all users of the machine. An application that is installed just for the current user should write the environment variables to the User environment space. The issue is how to accomplish this when environment variables are tied to a component and there is no way to condition the writing of these variables to different environment spaces depending on installation type.

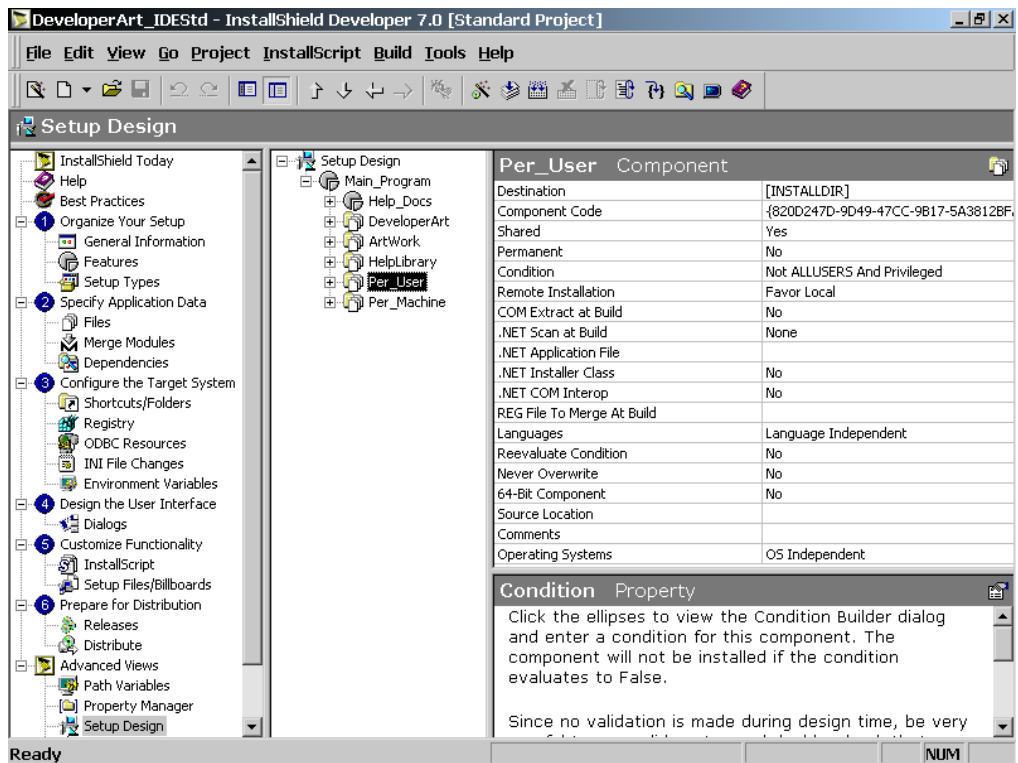


Figure 10-26: Special components in the Developer Art project for creating environment variables.

The answer is to create two special components whose only function is to create the environment variables in the appropriate environment space. You then condition each of the components so only one of them will be installed.

To implement a simple example of this approach, do the following:

1. First, create two special components under the `Main_Program` feature in the Setup Design view. Name these two components `Per_User` and `Per_Machine`, as shown in Figure 10-26.
2. Set the condition for each of these components As shown below:

Per_User component: (Not ALLUSERS) And Privileged

Per_Machine component: ALLUSERS And Privileged

The key to the properly implementing this approach is the condition that you place on these two special components. The `ALLUSERS` property determines where the configuration information for an application is stored. When the `ALLUSERS` property has a value, the configuration information is stored in the "All Users" profile. When the `ALLUSERS` property does not exist, the configuration information is stored in the user's personal profile. The `Privileged` property is set if an install is performed with elevated privileges or the user has administrative privileges.

3. In the Environmental Variables view, create an `EnvVar` environment variable twice as shown in Figure 10-27.
4. For the instance that is associated with the `Per_User` component, set the `Type` property to `User`.
5. For the instance associated with the `Per_Machine` component, set the `Type` property to `System`.
6. For each instance you give the same string as the value of the `Value` property. In this example, use the string `"Value1"`. Leave the default values for all of the other properties for each instance of the environment variable.

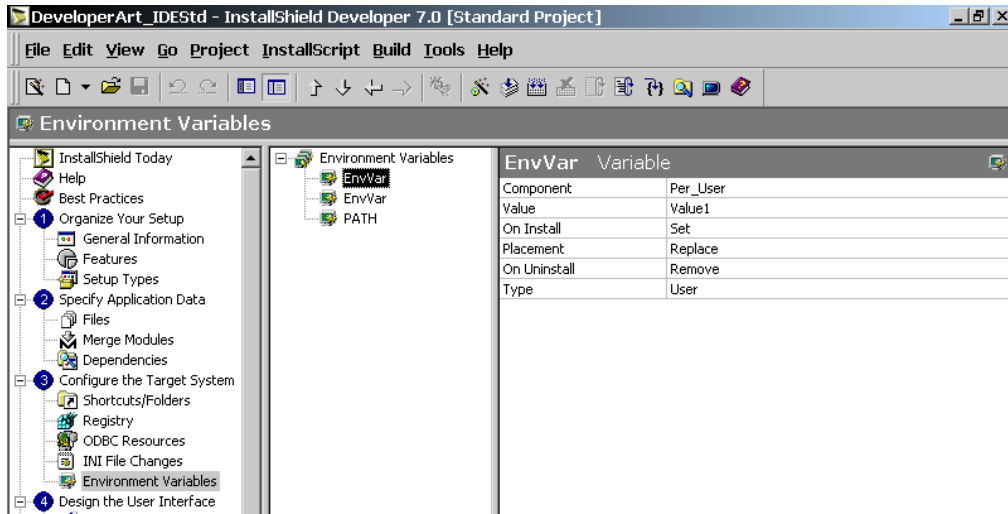


Figure 10-27: *Defining the environment variable for the two special components.*

To test, build the project and run the installation.

1. The first time through the installation, leave the default selection in the Customer Information dialog, which is to install the application for all users of the machine.
2. When the installation is complete, check to verify that the environment variable was written to the System environment space.
3. Uninstall and run the other option in the Customer Information dialog to see that the environment variable has now been written to the User environment space.

You now know how to set the value of environment variables using the Environment table. To complete our discussion of environment variables, the next section looks at how to access environment variables during the installation.

Accessing Environment Variables During an Installation

Obtaining the value of an environment variable that has been set prior to running an installation requires some InstallScript programming. For a Standard project, you can incorporate this programming in one of the event handlers if you want to obtain the value in the user interface sequence. You can use custom actions in both sequences of a Basic MSI project or in the execute sequence table of a Standard project. Remember that it is possible for different processes to have different environment variables.

InstallScript has a built-in function that allows you to obtain the value of an environment variable. A typical call to this function is as follows:

```
iResult = GetEnvVar("SYSTEMROOT", svValue);
```

The value of the SYSTEMROOT environment variable is returned by this function in the svValue argument. This function returns the value of the environment variable as it is set in the current process's environment block. Remember that environment variables defined for the user replace the value for the system when there is a conflict. A typical example of this is the TEMP environment variable, which has different values in the System and in the User environment space. Also remember that the PATH environment variable is cumulative, the value defined for the System and the value for the User are concatenated using the semi-colon delimiter.

More functionality is available if you use the Environment property available from the Windows Script Host. Chapter 9 covered this during the discussion about creating COM objects in InstallScript. Figure 10-28 shows some code for the OnBegin event handler where you use COM and the Windows Script Host object to get both values of the TEMP environment variable, as well as set the value of an environment variable that will exist only as long as the client process is alive.

The code shown in Figure 10-28 is available on the CD-ROM at the back of the book. You can copy the code out of the file named Figure 10-28.rul and paste it into Setup.rul replacing the code that is already there.

```

////////////////////////////////////
//
//  FUNCTION:   OnBegin
//
//  EVENT:     Begin event is always sent as the first event
//             during installation.
//
////////////////////////////////////
function OnBegin()
STRING  szUserValue, szSystemValue, szProcessValue1, szProcessValue2;
OBJECT  objShell, objSystemEnv, objUserEnv, objProcessEnv;
begin

    try
        // Create a Windows Script Host shell object.
        set objShell = CreateObject("WScript.Shell");

        // Create a collection of the environment variables
        // in the User environment space.
        set objUserEnv = objShell.Environment("User");

        // Create a collection of the environment variables
        // in the System environment space.
        set objSystemEnv = objShell.Environment("System");

        // Create a collection of the environment variables
        // in the Process environment space.
        set objProcessEnv = objShell.Environment("Process");

        // Get the value of the TEMP environment variable as
        // define in the User environment space. Need to expand
        // the environment string into an actual path value.
        szUserValue =
            objShell.ExpandEnvironmentStrings(objUserEnv("TEMP"));

        // Get the value of the TEMP environment variable as
        // define in the System environment space. Need to expand
        // the environment string into an actual path value.
        szSystemValue =
            objShell.ExpandEnvironmentStrings(objSystemEnv("TEMP"));

        // Display the value of both TEMP environment variables.
        sprintfBox(MB_OK, "Feedback",
            "TEMP value in the User space: %s\n\n" +
            "TEMP value in the System space: %s", szUserValue,
            szSystemValue);
    
```

Figure 10-28: *Accessing environment variables during an installation.*

```

// Get the value of the PATH environment variable in the
// Process environment space before changing it.
szProcessValue1 = objProcessEnv("PATH");

// Set the PATH environment variable with a temporary value.
objProcessEnv("PATH") = "C:\\MySetups";

// Get the value of the PATH environment variable in the
// Process environment space.
szProcessValue2 = objProcessEnv("PATH");

// Display the value of both PATH environment variables.
sprintfBox(MB_OK, "Feedback",
    "PATH value before change: %s\n\n" +
    "PATH value after change: %s", szProcessValue1,
    szProcessValue2);

catch
    sprintfBox(MB_OK, "Feedback", "Exception was thrown.");
endcatch;

end;
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
// FUNCTION:    OnEnd
//
// EVENT:      End event is the last event. It is not sent if the
//             installation has been aborted. In this case the
//             Abort event is sent
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
function OnEnd()
STRING  szProcessValue;
OBJECT  objShell, objProcessEnv;
Begin

    try
        // Create a Windows Script Host shell object.
        set objShell = CreateObject("WScript.Shell");

        // Create a collection of the environment variables
        // in the Process environment space.
        set objProcessEnv = objShell.Environment("Process");

        // Get the value of the PATH environment variable in the
        // Process environment space.
        szProcessValue = objProcessEnv("PATH");

```

Figure 10-28: *Continued.*

```

        // Display the value of the PATH environment variable.
        sprintfBox(MB_OK, "Feedback", "PATH value: %s",
                  szProcessValue);

    catch
        sprintfBox(MB_OK, "Feedback", "Exception was thrown.");
    endcatch;
end;

```

Figure 10-28: *Continued.*

The code shown in Figure 10-28 is placed in the `OnBegin` and `OnEnd` event handlers. In this way this same code can be easily used in a Basic MSI project, in addition to the Standard project you are using for this example.

The first thing that this code does is to create a Windows Host Script shell object. It also creates three environment collection objects, one each for the User, System, and Process environment variables. Using the User collection, the example code sets a variable to the value of the `TEMP` environment variable and then, using the System collection, sets another variable to the value of the `TEMP` environment variable. When these two string variables are displayed, you see that the values are different. The code then uses the `ExpandEnvironmentStrings` method of the shell object to convert the paths to absolute paths for the target machine. If this method was not used, the paths for the `TEMP` environment variable would include the `%SystemRoot%` and `%USERPROFILE%` strings instead of the actual folder names.

The code in Figure 10-28 demonstrates that if you set the value of an existing environment variable, the value is valid only in the present process and the new value does not exist after the process is shut down. This demonstration consists of setting the `PATH` environment variable to a new value and then displaying the value of the environment variable before and after it is set in the installation. You also get and display the value of the `PATH` environment variable in the `OnEnd` event handler. You see that the value as seen from the `OnEnd` event handler is the value to which the variable was changed: `C:\MySetups`. After the installation has completed, you can go and see that the change made to the `PATH` environment variable was not permanent.

The next section discusses how to work with initialization files during an installation.

Creating, Modifying, and Reading Initialization Files

The creation of initialization files and entries is another common operation that is implemented during an installation. Initialization files tend to contain information that is of interest only to the application itself and is less global in nature than entries made in the registry or environment variables set during an installation.

As with the creation of registry entries and the creation of environment variables, there is a special table in the Windows Installer database schema that is devoted to defining the entries to be made in initialization files. InstallShield Developer provides the INI File Changes view to assist with authoring this table. The INI File Changes view is available under Step 3 in the View List.

Click on the INI File Changes icon to see the INI File Changes view (Figure 10-29).

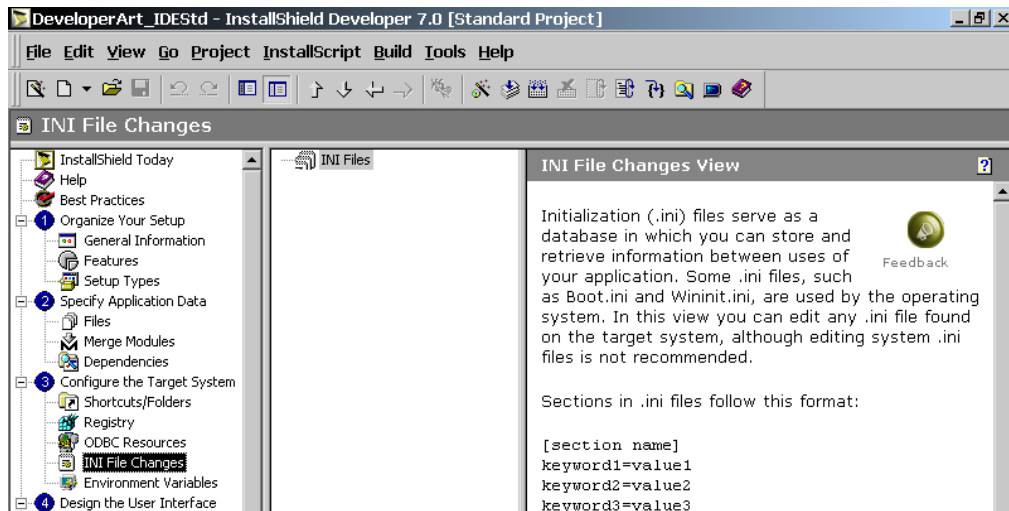


Figure 10-29: *The sub-view tree of the INI File changes view.*

In this sub-view tree you define the file names, sections, and keywords that are to be manipulated during the installation. To do this:

1. Right-click on the INI Files icon and select Add INI File.
2. Once you have defined a file name, right-click on the new INI file and select Add Section.
3. After defining a section, right-click on the section and select Add Keyword to define a keyword against which you will create a value.
4. Click on the file name, the section name, or the keyword to see the properties that you need to complete for each one.

Before we cover the required properties, we will look at the associated tables that are being authored.

The IniFile and RemoveIniFile Tables

To create or modify an initialization file during an installation, you need to author the IniFile table. You should use this table to create your initialization file entries instead of trying to programmatically create them using InstallScript code. Using the IniFile table and the RemoveIniFile table takes full advantage of the capabilities for performing an installation using elevated privileges. The schema of the IniFile table is described in Table 10-8.

Table 10-8: The IniFile and RemoveIniFile Table Schemas

Column Name	Description
IniFile or RemoveIniFile	This column is the primary key for the table and it needs to conform to the requirements of the Identifier data type.
FileName	This specifies the name of the INI file that is to be created or modified.
DirProperty	This column contains the name of a property that resolves to an absolute path to the folder in which the INI file is to be created or found. This column can be NULL and, if that is the case, the Windows Installer will look for or create the INI file in the Windows folder (%Windows%).

Table 10-8: The IniFile and RemoveIniFile Table Schemas

Column Name	Description
Section	This column specifies the name of the section. This name needs to conform to the requirements of the Formatted data type. This means that you can use property names inside square brackets as part of the name being specified.
Key	This column specifies the name of the key inside the section specified in the previous column. This name needs to conform to the requirements of the Formatted data type.
Value	This column specifies the name of the value to be written against the key specified in the previous column. This name needs to conform to the requirements of the Formatted data type.
Action	This column contains an integer value that specifies the type of operation that is to be taken on the INI file. There are five possible types of operations, as defined in Table 10-9.

The purpose of the IniFile table is to define INI files that are to be created and values that are to be added to an INI file when the associated component is installed. The purpose of the RemoveIniFile table is to define INI file entries that are to be deleted during an installation. The WriteIniValues standard action reads the IniFile table and performs the specified operations. The RemoveIniValues standard action reads the RemoveIniFile table and performs the specified deletions from the specified INI files. This standard action also removes any values that were created during the installation of a component when that component is uninstalled.

In the Action column, the type of operation that is to be taken on the INI file is specified by an integer value. The description of these values is provided in Table 10-9.

Table 10-9: Valid Values for the Action Column

Value	Meaning
0	Specifies that the indicated INI file value will be created or updated. This replaces the value of a key if the key already exists in the INI file. This value is valid only for the IniFile table.
1	Specifies that the indicated INI file value will be created only if the key does not already exist. If the key already exists, the current value is not replaced. This value is valid only for the IniFile table.
2	Specifies that the indicated key and its value will be removed from the INI file when the associated component is installed. This value is valid only for the RemoveIniFile table.
3	Specifies that the indicated INI file value will be created or appended to the current value if the key already exists. When appending to an existing value, the delimiter used is the comma (,). This value is valid only for the IniFile table.
4	Specifies that the indicated value will be removed from the INI file when the associated component is installed. This will remove the key only if the specified value is the only value for the key. This value is valid only for the RemoveIniFile table.

When you work in the INI File Changes view, you are creating rows in either the IniFile table or the RemoveIniFile table depending on the action you select for each keyword. You will see how this view works by creating an example in the next section.

Working in the INI File Changes View

In this section, you will work through a small example to get the feel for what can be done to create or modify INI files during an installation. The Developer Art application does not use an initialization file, so you have to create one for this example.

Before you get into an example lets first look at the properties that need to be set for both the initialization file and the keywords that you want to place in the file. The properties associated with the name of an initialization file are discussed in the following list.

Display Name: This property is used to populate the FileName column of either the IniFile or the RemoveIniFile tables. This property is populated when you first create the name of the INI file. If the file meets the 8.3 file-naming convention, this property displays only the name of the file. If the file name is more than eight characters and/or contains spaces, the display name will contain both the short file name and the long file name separated by a vertical bar. The name used for the INI file is associated with a default sting ID so it can be localized if necessary.

Component: This property contains the name of the associated component. If this component is installed, the indicated operations are performed on the initialization file. The value in this field is used to populate the component_ column of the IniFile and the RemoveIniFile tables. When you click in this field an ellipsis button displayed. This ellipsis button launches a dialog from which you can pick the component with which you want to associate the initialization file.

Target: The value for this property defines the folder where the initialization file is to be created or found. The entry for this property needs to be a property name inside square brackets and this property must resolve to an absolute location on the target machine. This value is used to populate the DirProperty column of the IniFile and RemoveIniFile tables. The default value for this property is [INSTALLDIR]. You can click in this field to get a drop-down combo box that provides a selection of the operating system defined locations. You can select the “Browse, create, or modify a directory entry” option to define a directory-related property name. This option was discussed in Chapter 2 and the resolution of the Directory table was discussed in Chapter 3. When this property is set to NULL

the Windows Installer will use the %Windows% folder as the location for the initialization file.

The only property for a section name is the name of the section itself. The only purpose of having a property for the section name is to allow you to have access to the string table so that this name can be localized. There are three properties for a keyword as shown Figure 10-30.

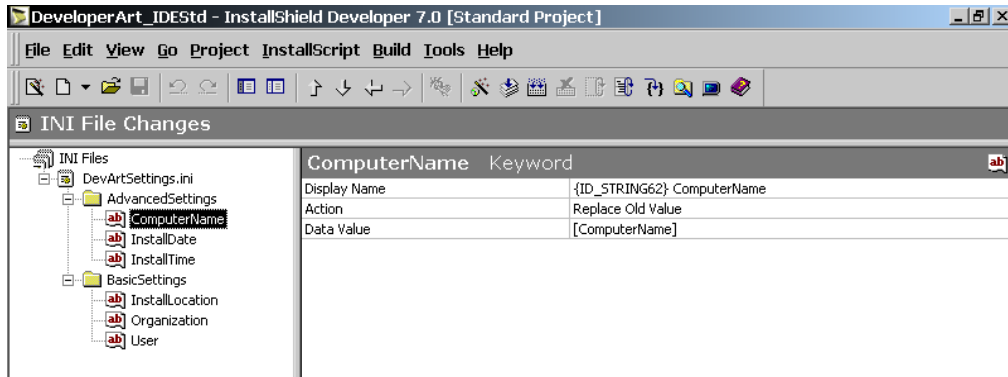


Figure 10-30: *The property set for a keyword.*

Each of these properties is discussed in the following list.

Display Name: This property is used to populate the Key column of the IniFile and the RemoveIniFile tables. This property is populated when you first create the name of the keyword. The name used for the keyword is associated with a default sting ID so that it can be localized if needed. You do not need to do anything with this property.

Action: This property provides a drop-down menu that provides a selection of the values that can be placed in the Action column of the IniFile or the RemoveIniFile table. The selections available in this menu are as follows:

- **Replace Old Value:** This option replaces an existing keyword and its value with the value that you place in the Data Value property. This option places a 0 in the Action column of the IniFile table.
- **Do Not Overwrite:** This option prevents an existing keyword and its value from being replaced with the value that you place in the

Data Value property. This option places a 1 in the Action column of the IniFile table.

- ***Remove Whole Value:*** This option removes an existing keyword and its value during an installation. This option places a 2 in the Action column of the RemoveIniFile table.
- ***Append Tag:*** This option creates the keyword if it does not exist or appends to the value if the keyword does exist. This option places a 3 in the Action column of the IniFile table.
- ***Remove Tag:*** This option removes the value from the keyword during an installation. This option places a 4 in the Action column of the RemoveIniFile table.

Data Value: The entry in this property is used to populate the Value column of the IniFile or the RemoveIniFile table. This property takes a formatted text string, which means that you can use property names in square brackets to write the information that you want as the data for keyword.

In this example, you will create an initialization file during an installation that is not removed when the application is uninstalled. This way, you can experiment with some of the modes of working with existing files. To generate a situation where an initialization file is created during an installation, but not removed during an uninstallation you can create a special component with a NULL value for the Component Code property. In this fashion the Windows Installer will not know about this component and it will not get uninstalled.

1. Create this component under the MainProgram feature as shown in Figure 10-31 and name this component INI_Component.
2. Remove the default GUID from the Component Code property. This makes this component invisible to the Windows Installer, so the associated INI file will not be uninstalled.

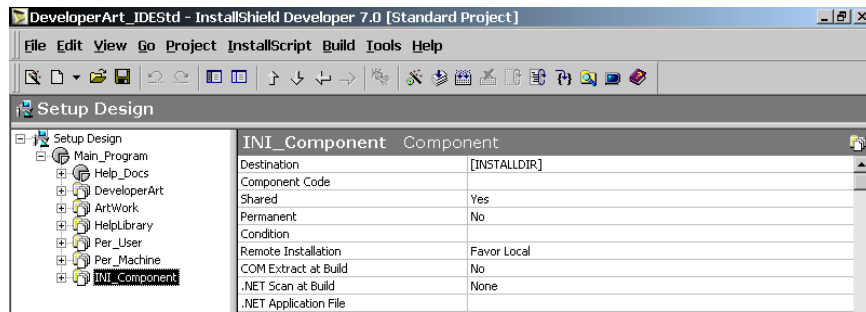


Figure 10-31: Special component for creating a permanent initialization file.

- Click on the INI File Changes view and define an initialization file by right clicking on the INI Files node and selecting the Add INI File option. Name this INI file DevArtSettings.ini as shown in Figure 10-32.
- Select this INI file to display the property panel and then select the INI_Component for the value in the Component field. Also delete the value in the Target property. With this value NULL, the Windows Installer assumes that the initialization file is to be located in the Windows folder.

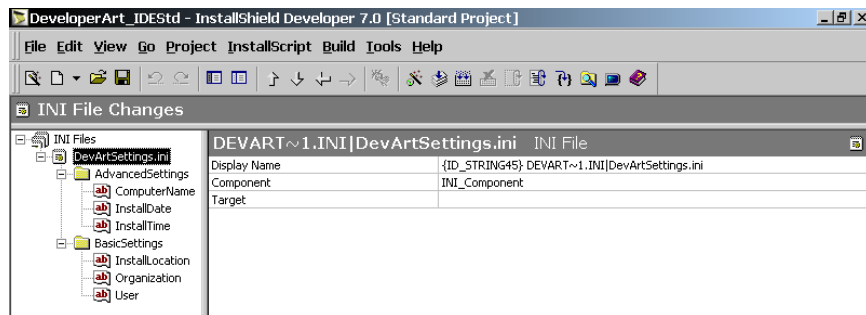


Figure 10-32: The properties panel for the name of the initialization file to be manipulated.

- Add two sections in the INI file and under each of these sections add three keywords as shown in Figure 10-32. When you click on one of the section names, you see that there is only one property, called Display Name. This property is the name of the section and it is associated with a string ID that allows you to localize this name if necessary. The value of

this property is used to populate the Section column of the IniFile and RemoveIniFile tables. You do not need to do anything with this property.

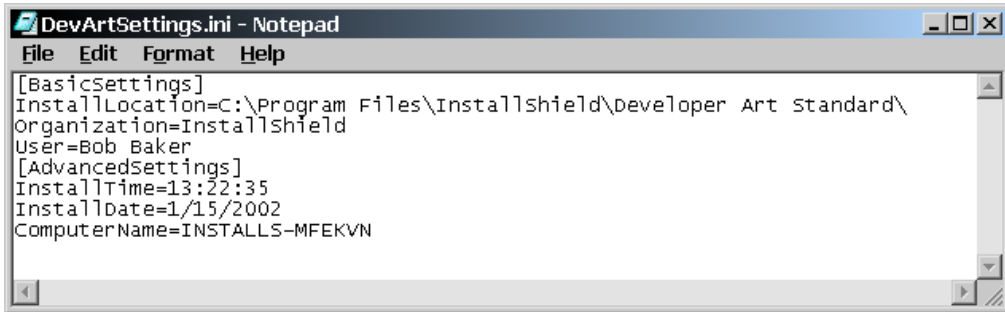
- For each keyword you need to set the value of the Action and the Data Value properties. The entries that you should make are shown in Table 10-10

Table 10-10: Keyword Actions and Values

Keyword	Action	Data Value
ComputerName	Replace Old Value	[ComputerName]
InstallDate	Append Tag	[Date]
InstallTime	Append Tag	[Time]
InstallLocation	Replace Old Value	[INSTALLDIR]
Organization	Replace Old Value	[COMPANYNAME]
User	Replace Old Value	[USERNAME]

With this set of actions and values, you will be able to build a list of the dates and times that you installed the Developer Art application. Before testing this example you might want to comment out the OnBegin and the OnEnd event handlers in your project to avoid all the message boxes that are displayed showing the value of different environment variables. All you need to do is then build the project and install it.

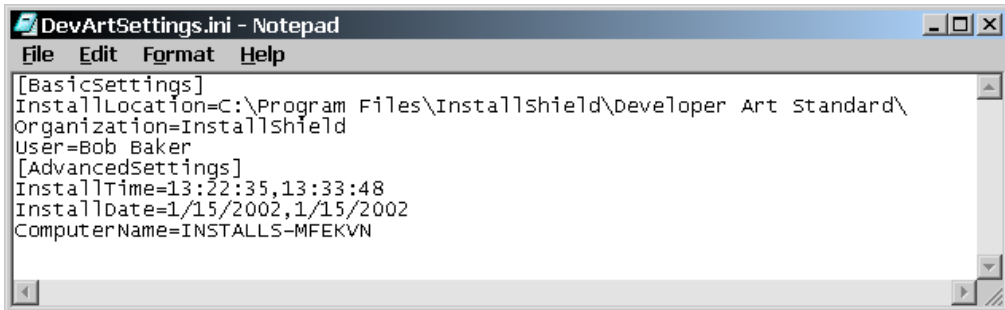
After the installation is complete, look in the Windows folder to find the initialization file that was created. When you open this file, you should see entries that are similar to those that are shown in Figure 10-33.

A screenshot of a Notepad window titled "DevArtSettings.ini - Notepad". The window has a menu bar with "File", "Edit", "Format", and "Help". The text content is as follows:

```
[BasicSettings]
InstallLocation=C:\Program Files\InstallShield\Developer Art Standard\
Organization=InstallShield
User=Bob Baker
[AdvancedSettings]
InstallTime=13:22:35
InstallDate=1/15/2002
ComputerName=INSTALLS-MFEKVN
```

Figure 10-33: *The initial contents of the DevArtSettings.ini file after the first installation.*

To see the impact on this initialization file of a second installation, you need to uninstall the Developer Art application and notice that the DevArtSettings.ini file is not removed. Then reinstall the application and look at the contents of this initialization file. You should see something that looks like what is shown in Figure 10-34. You now have appended the date and time of the second installation to the values for the first installation.

A screenshot of a Notepad window titled "DevArtSettings.ini - Notepad". The window has a menu bar with "File", "Edit", "Format", and "Help". The text content is as follows:

```
[BasicSettings]
InstallLocation=C:\Program Files\InstallShield\Developer Art Standard\
Organization=InstallShield
User=Bob Baker
[AdvancedSettings]
InstallTime=13:22:35,13:33:48
InstallDate=1/15/2002,1/15/2002
ComputerName=INSTALLS-MFEKVN
```

Figure 10-34: *The impact of a second install of the Developer Art application.*

Reading Initialization Files During an Installation

One of the methods to access the values of an existing initialization file is by writing InstallScript code. InstallScript provides a number of built-in functions that work with initialization files. Three of these functions are used to read values in an initialization. These three functions are GetProfInt, GetProfString, and GetProfStringList. These functions use the Windows API private profile functions.

In this section you will create some functionality to read the values from a section in an initialization file and print the keyword and values to a text file. You will create this functionality in the OnBegin event handler. This way you can use the same code in a Basic MSI project to perform the same operation without having to make any changes. This example uses the GetProfStringList function to get a list of all the functions that are in the [FuncWiz - Category - All] section of the Funcwiz.ini file. You created a modified version of this file in Chapter 8 and you need to copy this file to the Windows folder before beginning this example.

The code to access the Funcwiz.ini file in the Windows folder and to print out the keywords and values under the [FuncWiz - Category - All] section is shown in Figure 10-35.

```

////////////////////////////////////
//
//  FUNCTION:   OnBegin
//
//  EVENT:      Begin event is always sent as the first event
//              during installation.
//
////////////////////////////////////
function OnBegin()
OBJECT  fso, file, tempfldr, dictionary;
LIST    Key, Value;
STRING  szFileName, szKey, szValue, szLine, szKeys();
INT     i, iCnt, iReturn;
VARIANT DictItem;
begin

```

Figure 10-35: *InstallScript code for accessing an initialization file and printing the values.*

```

// Set the name of the text file to be created.
szFileName = "FuncWiz Functions.txt";

// Create the two lists that will hold the
// keyword and value pairs.
Key = ListCreate(STRINGLIST);
Value = ListCreate(STRINGLIST);

// Read the keywords and values for the
// "FuncWiz - Category - All" section in the Funcwiz.ini file.
// Since the file is in the Windows folder you do not need to
// include a path as part of the first argument.
GetProfStringList("Funcwiz.ini",
                  "FuncWiz - Category - All", Key, Value);

try
    // Create a dictionary object hold the keyword/value pairs.
    set dictionary = CreateObject("Scripting.Dictionary");

    // Initialize the two lists for traversing.
    iReturn = ListSetIndex(Key, LISTFIRST);
    ListSetIndex(Value, LISTFIRST);

    // Loop through the two lists and place their values
    // into the dictionary object.
    while(iReturn != END_OF_LIST)
        ListCurrentString(Key, szKey);
        ListCurrentString(Value, szValue);

        // Values are added to dictionary object
        // using the Add method.
        dictionary.Add(szKey, szValue);

        iReturn = ListSetIndex(Key, LISTNEXT);
        ListSetIndex(Value, LISTNEXT);
    endwhile;

    // Destroy the two lists now that you
    // do not need them any longer.
    ListDestroy(Key);
    ListDestroy(Value);
Catch
    SprintfBox(MB_OK, "Feedback",
              "Exception when accessing the dictionary object");
endcatch;

```

Figure 10-35: *Continued.*

```

try
    // Create a FileSystemObject that you will use to
    // generate the text file to which you will write
    // the keywords and values from the initialization file.
    set fso = CreateObject("Scripting.FileSystemObject");

    // Get the location of the temp folder.
    set tempfldr = fso.GetSpecialFolder(2);

    // Create the absolute path to the text file
    // that you will use for writing and then
    // create the text file.
    szFileName = tempfldr ^ szFileName;
    set file = fso.CreateTextFile(szFileName, TRUE);

    // Get the number of elements in the
    // dictionary object and set the szKeys
    // array to be that size.
    iCnt = dictionary.Count;
    Resize(szKeys, iCnt);
    szKeys = dictionary.Keys();
    // Loop through the dictionary object and
    // write the keywords and the values to the
    // text file with the values delimited by a tab.
    for i=0 to iCnt-1
        DictItem = szKeys(i);
        szLine = szKeys(i) + "\t" + dictionary.Item(DictItem);
        file.WriteLine(szLine);
    endfor;

    // Close the text file.
    file.Close();

catch
    sprintfBox(MB_OK, "Feedback",
        "Exception when accessing the FileSystemObject object");
endcatch;

end;

```

Figure 10-35: *Continued.*

In this code example, the main process is to access the initialization file from the Windows folder and to use the `GetProfStringList InstallScript` function to create two lists, one for the keywords and one for the values. The code takes the contents of these two lists and creates a dictionary object out of the values in the lists. It then destroys the lists because it no longer needs them. Because it is an associative

array, a dictionary object is ideal for holding the contents of the keywords and their values.

Once you have the dictionary object, you get the count of the elements in this object and then loop through the object to write each keyword/value pair to a text file. When finished writing the lines to the text file, the example closes the file. The contents of one section of the Funcwiz.ini file are captured and written to a text file. During an installation you probably will not need to write the values in an initialization file to another text file but this example shows how to access an initialization file. One of the things that could be done is to read an initialization file and set a property to one of the values in this file. If it were a public property then the value would be available in both the UI sequence and the installation sequence.

The next topic explains how to search for existing files, folders, and registry entries.

Searching for Files, Folders, and Registry Entries

One of the most common operations that you need to implement in an installation is to search for entities that may already be installed on the target machine. Most often you search for a file that will indicate that a certain application is already installed. However, you may also want to search for the existence of a folder, a registry entry, and/or an entry in an initialization file.

There are many reasons that you may want to search the target system prior to running your installation. It is possible that your application needs to have another application already installed or you might want to find the install location of an earlier version of your application so you can install an upgrade to the same location. Another slightly different reason for searching the target machine for the existence of a particular file is to see if the machine is in compliance with the installation requirements for the purchased software. Compliance checking is done, for example, when you sell a competitive upgrade at a lower price with the stipulation that the end user must own the competing product.

The Windows Installer provides several standard actions and tables that are used to implement searches of the target system. Searching the target system can be implemented in InstallScript, but you are encouraged to use the built-in Windows Installer functionality.

This section emphasizes how to manipulate the Windows Installer capabilities for performing searches. At the end of this section, you will take a look at the InstallScript event handlers that can be used in a Standard project to perform script-based searches.

The first subject is the mechanism that is used by the Windows Installer to implement the searching for files, folders, registry, and initialization file entries.

How the Basic Search Mechanism Works

One of the standard actions in both the InstallUISequence table and the InstallExecuteSequence table is the AppSearch action. To search for a file or folder, this action reads a number of tables which you need to author if you want a search performed. The six tables that are involved in performing searches are the AppSearch, Signature, CompLocator, RegLocator, IniLocator, and DrLocator tables. The schema of these six tables is shown in Figure 10-36.

Not all of the tables shown in Figure 10-36 are used to implement any particular search. The AppSearch action reads the AppSearch table and then starts the search process. If the search process is successful, the name of the property in the first column of the AppSearch table is set to the results of the search. The property can then be used in a condition or it can be queried for its value, which then can be used as required. If the search is unsuccessful, the property has a NULL value.

The search logic that is used is outlined in Figure 10-37. This diagram shows that the locator tables are read in a specific order. This means that you have to be careful in how you author these tables to make sure you are getting the desired search results.

As shown in the diagram in Figure 10-37, there are two branches to this logic. If the signature is identified in the Signature table, the search is for a file. Otherwise, the search is for a folder, initialization file entry, or a registry entry. The locator tables provide a starting point for performing the search. When the search for a particular signature in the AppSearch table is successful, the associated property is set to the

results of the search. If the search is not successful, the property keeps its original value, which in most cases is NULL.

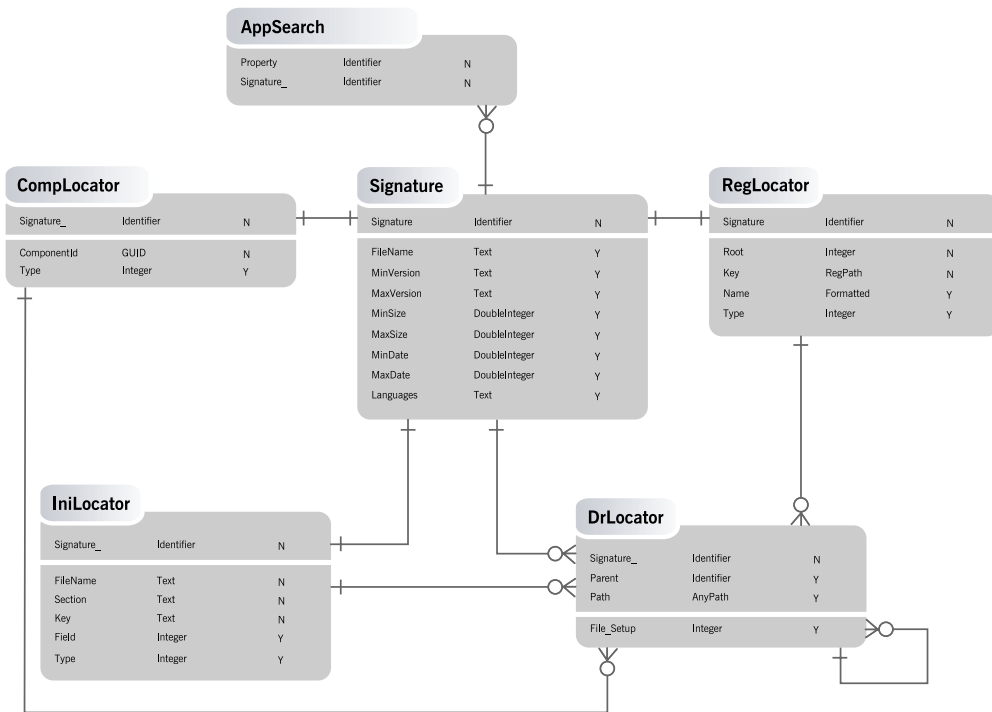


Figure 10-36: The schema of the tables used to implement searches of a target system.

There is a System Search view under Step 5 in InstallShield Developer that provides an easy approach to defining a search. In this chapter you will use the Direct Editor under Advanced Views and make entries directly in the affected tables. This will acquaint you with the details of how searching is accomplished. Using this approach you have to know the proper entries to make in each column of the six tables that are shown in Figure 10-36. The following sections examine each of these tables.

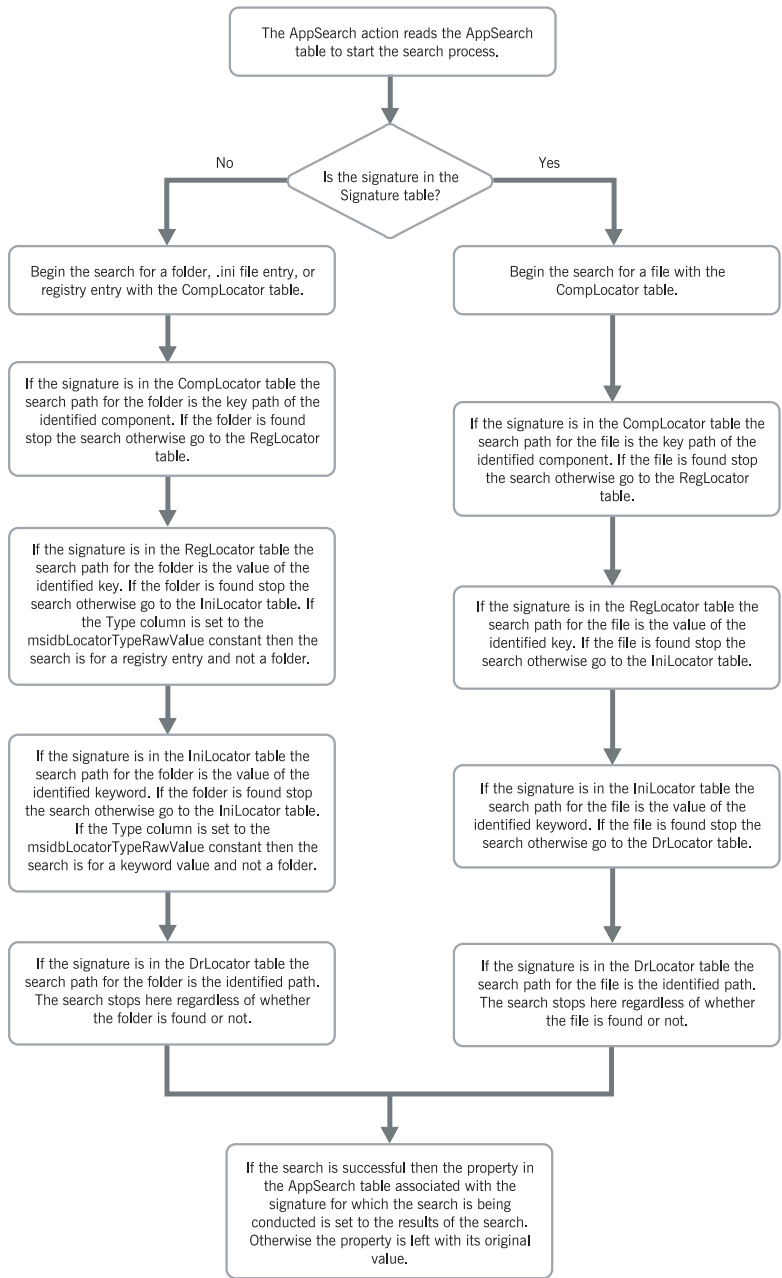


Figure 10-37: *The basic search logic implemented by the Windows Installer.*

The AppSearch Table

The AppSearch table is the starting point for all searches that you want to capture the results as the value of a property. This table's two columns are described in Table 10-11.

Table 10-11: The AppSearch Table Schema

Column Name	Description
Property	This is the name of the property that will be set to the results of the search for the signature identified in the next column. This property can be initialized in the Property table or from the command line. If the search is successful, the initial value is overridden. Normally this property is a public property so that its value can be passed to the process running the actions in the InstallExecuteSequence table.
Signature_	This value is a unique identifier that is a foreign key into the first column of the other five tables that are used by the Windows Installer to implement searching.

Even though the AppSearch table starts the search process there can be other searches initiated as part of the process of searching for the signature in this table. The results of these other searches are not saved as the value of a property unless the associated signature is also entered into the AppSearch table. You will see an example of this later in this chapter.

The Signature Table

The purpose of the Signature table is to identify all the attributes of a file for which you want to search. If you do not want to search for a file, leave this table empty. The nine columns in the Signature table are described in Table 10-12.

Table 10-12: The Signature Table Schema

Column Name	Description
Signature	This value is the primary key and identifies a particular file signature for which a search is to be conducted.
FileName	The name of the file for which a search is to be conducted.
MinVersion	The minimum version of the file for which the search is being conducted.
MaxVersion	The maximum version of the file for which the search is being conducted.
MinSize	The minimum size of the file for which the search is being conducted.
MaxSize	The maximum size of the file for which the search is being conducted.
MinDate	The minimum creation date of the file for which the search is being conducted. This date needs to be provided in the MS-DOS format as follows: Bits 0–4: Day of the month Bits 5-8: Month where January is 1 Bits 9-15: Year offset from 1980
MaxDate	The maximum creation date of the file for which the search is being conducted. This date needs to be provided in the MS-DOS format.
Languages	A delimited list of the languages supported by the file for which the search is being conducted.

Normally a file search requires that only the first two columns be populated. The remaining columns are necessary only if the search has to distinguish between numbers of files with the same name.

The CompLocator Table

The CompLocator table is the first of what the *locator* tables to be accessed during the search process. The purpose of this table is to define the starting location of a search in terms of a component's configuration data. The three columns in the CompLocator table are described in Table 10-13.

Table 10-13: the CompLocator Table Schema

Column Name	Description
Signature_	The primary key for this table and a foreign key into the Signature table if the search is being conducted for a file. It is through this value that a search is how the link is made with the property defined in the AppSearch table.
ComponentId	The GUID that identifies component in the registry.
Type	An integer value that defines whether the key path for the component is a file name or a folder. If a value of 0 is used in this column, the key path is a folder. If a value of 1 is used, the key path is a file name. If this column is NULL, the value is defaulted to 1.

When a component is installed, its component ID is written to the registry with a value name of the product code GUID. The value data for this value name is the key path for the component. There can be four different types of key paths: a file name, the folder in which the component is installed, a registry entry, and a reference to an ODBC data source name associated with the component. The two types of key paths that you are interested in here are the file name key path and the folder key path.

The RegLocator Table

You can use the RegLocator table to focus your search for a file or a folder and you can also use it to obtain a raw value from the registry that has nothing to do with the location of a file or a folder. This third possibility relieves you of having to write code in order to extract values from the registry. The five columns in the RegLocator table are described in Table 10-14.

Table 10-14: The RegLocator Table Schema

Column Name	Description
Signature_	The primary key for this table and a foreign key into the Signature table if the search is being conducted for a file.
Root	The integer value placed in this column indicates the root key under which the branch in the next column exists. The acceptable values are the same as shown in Table 10-2 except that the -1 value is not used here.
Key	This is the branch in the registry under the root defined in the previous column. This value cannot have a leading or an ending backslash (\).
Name	This column contains the value name that has the data value that you want to use. If this column is NULL, it is assumed that it is the default value that is to be used.
Type	The value in this column determines if it is a file, folder or a raw registry value for which you are searching. The permissible values are as follows: <ul style="list-style-type: none"> Value = 0: The search is for a folder. Value = 1: The search is for a file. Value = 2: The search is for a raw registry value.

One of the most helpful keys in the registry that you can use in a search process is the App Paths key. This is the key to which you write when you create an application path. If you are targeting a 64-bit operating system and you want to search the 64-bit registry instead of the 32-bit registry, you would add 16 to the acceptable values for the Type column shown in Table 10-14.

The IniLocator Table

The operation of the IniLocator table is much like the RegLocator table. You can use the IniLocator table to focus your search for a file or a folder and you can also use it to obtain a raw value from an initialization file that has nothing to do with the location of a file or a folder. This third possibility relieves you of having to write code in order to extract values from an initialization file. The six columns in the IniLocator table are described in Table 10-15.

Table 10-15: The IniLocator Table Schema

Column Name	Description
Signature_	The primary key for this table and a foreign key into the Signature table if the search is being conducted for a file.
FileName	The name of the initialization file that the search process will access.
Section	The name of the section in the initialization file that contains the keyword of interest.
Key	The name of the keyword in the section defined in the previous column.
Field	The value field to be extracted from the keyword. This is applicable if the value has more than one value delimited by commas. Field numbers start with 1. If the value in this field is NULL or 0 then the complete value is retrieved.

Table 10-15: The IniLocator Table Schema (Continued)

Column Name	Description
Type	<p>The value in this column determines if it is a file, folder or a raw keyword value for which you are searching. The permissible values are as follows:</p> <p>Value = 0: The search is for a folder.</p> <p>Value = 1: The search is for a file.</p> <p>Value = 2: The search is for a raw keyword value.</p>

Note that there is no column here for defining the location of the initialization file that you are using in the search. This is because the initialization file can only be in the Windows folder.

The DrLocator Table

The DrLocator table is where you explicitly define the directory location on the target machine that you want to search. In this table you can reference the other locator tables as part of the definition of a search location. The four columns of the DrLocator table are described in Table 10-16.

Table 10-16: The DrLocator Table Schema

Column Name	Description
Signature_	Part of the primary key for this table and a foreign key into the Signature table if the search is being conducted for a file.

Table 10-16: The DrLocator Table Schema (Continued)

Column Name	Description
Parent	Part of the primary key for this table and a foreign key into the locator tables including this table and into the Signature table. This key is the signature of the parent of the file or directory that is defined by the signature in the previous column.
Path	The value in this column is that path on the target system where the search is to be conducted. This can be a relative path under the path defined by the signature in the Parent column or it can be an absolute path. You can use properties inside square brackets to define this path value.
Depth	In this column you place the depth of the search below the path defined in the previous column. A value of 0 means that you will search only the directory defined in the previous column.

If you leave both the Parent and the Path columns NULL for a particular signature you will search all the fixed drives of the target system for a file or folder. The fact that the Parent column holds the signature of an entry in one of the locator tables or into the Signature table allows you to perform searches, the results of which can be used in additional searches.

The next section provides some examples of how this searching works.

Basic Searching Examples

In the examples in this section, you will use the same Standard project you have been using in the first part of this chapter. Everything you do here works the same in a Basic MSI project. Before beginning these examples, you need to create a facility to display the results of the search. You do this by creating a small function that displays

the value of the property defined in the AppSearch table. The code for this function is shown in Figure 10-38.

```

prototype SearchFeedback();

/////////////////////////////////////////////////////////////////
//
//  FUNCTION:    SearchFeedback
//
//  PURPOSE:    The purpose of this function is to report the
//              results of a search of the target system.
//
/////////////////////////////////////////////////////////////////
function SearchFeedback()
STRING  szPropValue;
INT     iBufSize;
begin

    szPropValue = "";
    iBufSize = 0;

    MsiGetProperty(ISMSI_HANDLE, "SEARCH", szPropValue, iBufSize);

    iBufSize++;
    Resize(szPropValue, iBufSize);

    MsiGetProperty(ISMSI_HANDLE, "SEARCH", szPropValue, iBufSize);

    sprintfBox(MB_OK, "Feedback", "The search results are:\n%s",
                                                       szPropValue);

end;

```

Figure 10-38: *The SearchFeedback function for displaying search results.*

You can place the code anywhere in Setup.rul, but you will call it from inside the OnFirstUIBefore event handler. You can place the call to this function as the first executable statement inside this event handler. In the code shown in Figure 10-38, you should note that you will be using a public property named SEARCH to capture the results of the searches. To make the entries in the various tables, you will use the Direct Editor. Using the Direct Editor is just like using the Orca database editing utility.

Searching for a File

When defining a search for a file, you first need to discern if you can narrow the search or whether you need to scan the entire target system. First, assume that the file location is unknown, so you have to search the entire target system.

SEARCHING ALL FIXED DRIVES

Here you will search for the main executable of InstallShield Developer. The name of this file is `isdev.exe` and it is located in the root installation location for InstallShield Developer (though, for this example, you are assuming its location is unknown). Implementing a complete search of all fixed drives requires that you make one entry in each of the AppSearch, Signature, and DrLocator tables. These entries are shown in Table 10-17.

Table 10-17: Searching All Fixed Drives for a File

AppSearch Table Entries

Column Name	Value
Property	SEARCH
Signature_	Sig1

Signature Table Entries

Column Name	Value
Signature	Sig1
Filename	<code>isdev.exe</code>
Remaining columns	NULL

Table 10-17: Searching All Fixed Drives for a File (Continued)**DrLocator Table Entries**

Column Name	Value
Signature_	Sig1
Parent	NULL
Path	NULL
Depth	5

The only comment that needs to be made about this example is the value used for the Depth column in the DrLocator table. Since you do not know where the file is, you do not know how deep it is in any particular directory tree. If you left the depth at 0, only the root drives would be searched. Indicate a depth of 5 to be on the safe side. The first file that is found with the name of isdev.exe stops the search process and then the value of the SEARCH property is displayed in a message box.

SPECIFYING THE SEARCH PATH FOR A FILE

In this example, you will use the RegLocator table to provide a location to search for the isdev.exe file. In this example you have to create one row in each of the AppSearch, Signature, and RegLocator tables. The entries for this example are shown in Table 10-18.

Table 10-18: Specifying the Search Path for a File**AppSearch Table Entries**

Column Name	Value
Property	SEARCH
Signature_	Sig1

Table 10-18: Specifying the Search Path for a File (Continued)**Signature Table Entries**

Column Name	Value
Signature	Sig1
Filename	isdev.exe
Remaining columns	NULL

RegLocator Table Entries

Column Name	Value
Signature_	Sig1
Root	2
Key	SOFTWARE\Microsoft\Windows\CurrentVersion\App Paths\isdev.exe
Name	Path
Type	0

The only thing that you did differently in this example than when you were searching for the file on all fixed drives is to narrow the search using the RegLocator table instead of using the DrLocator table. You are able to pinpoint the location to search for the isdev.exe file by locating the value for the Path registry value for this file. As discussed earlier in this book, the Path registry value is the path where this file is installed.

The process that was followed for this search was first for the AppSearch action to note that the signature value is in the Signature table so that this makes it a search for a file. Secondly the AppSearch action checks each of the locator tables to see in which

the signature is also available and then uses the information in the locator table to narrow the search.

SEARCHING FOR A FILE IN A SPECIFIC LOCATION

In this example, you search for a file in a specific location and you also include a search for the location as part of the search process. You need to enter a row in each of the AppSearch, Signature, RegLocator, and DrLocator tables. The required values are shown in Table 10-19.

In this example, you reference a second signature that does not appear in the AppSearch or the signature tables. The Sig2 signature is defined in the RegLocator table and then used in the Parent column of the DrLocator table. Also in the DrLocator table you set the Depth column to 0 so that only the folder defined by the Sig2 signature will be searched.

Table 10-19: Searching a Specific Location for a File

AppSearch Table Entries

Column Name	Value
Property	SEARCH
Signature_	Sig1

Signature Table Entries

Column Name	Value
Signature	Sig1
Filename	isdev.exe
Remaining columns	NULL

Table 10-19: Searching a Specific Location for a File (Continued)**RegLocator Table Entries**

Column Name	Value
Signature_	Sig2
Root	2
Key	SOFTWARE\Microsoft\Windows\CurrentVersion\App Paths\isdev.exe
Name	Path
Type	0

DrLocator Table Entries

Column Name	Value
Signature_	Sig1
Parent	Sig2
Path	NULL
Depth	0

Note, in this example, that you have performed two searches, one for the folder where you want to find the file and another to see if the file is in that folder. This shows the flexibility of the searching mechanism implemented by Windows Installer.

Searching for a Folder

Here you will look at two examples of how to search for a folder. One of these examples searches for a file and then gets the folder in which that file resides. The other example uses an initialization file to retrieve the location of a particular folder.

GETTING THE FOLDER IN WHICH A FILE EXISTS

One of the files that is installed by InstallShield Developer is a merge module that is created by Microsoft. The name of this file is `ATL.MSM` and it is buried several layers deep under the installation root directory of InstallShield Developer. In this example, you are going to search for the file `ATL.MSM` and return in the `SEARCH` property the folder in which this file is located. To make this search work, you need to create two rows in the `DrLocator` table, as well as have one row each in the `AppSearch`, `Signature`, and `RegLocator` tables. The values that you want to enter in these four tables are shown in Table 10-20.

The complexity of this example comes in the use of three different signatures and the use of the `Parent` column in the `DrLocator` table to initiate searches that are not listed in the `AppSearch` table. What you are doing in the `DrLocator` table is to search for the folder that you want to find by specifying the signature of the file in the `Signature` table as the parent of the folder that you want to find. This forces the search mechanism to perform a search for this file, which it does by trying to resolve the second row in the `DrLocator` table. The second row specifies that the parent of the signature `Sig2` is the signature `Sig3`. The signature `Sig3` is then found to be defined in the `RegLocator` table, which retrieves the root install location for InstallShield Developer. The search then continues to search for the `ATL.MSM` file to a depth of 3 under the root location. When the file is found it can then set the `SEARCH` property to be the folder in which `ATL.MSM` is located.

Table 10-20: Searching for the Folder in Which a File is Located**AppSearch Table Entries**

Column Name	Value
Property	SEARCH
Signature_	Sig1

Signature Table Entries

Column Name	Value
Signature	Sig2
Filename	ATL.MSM
Remaining columns	NULL

DrLocator Table Entries

Column Name	Value
Signature_ (Row 1)	Sig1
Parent (Row 1)	Sig2
Path (Row 1)	NULL
Depth (Row 1)	NULL
Signature_ (Row 2)	Sig2
Parent (Row 2)	Sig3
Path (Row 2)	NULL
Depth (Row 2)	3

Table 10-20: Searching for the Folder in Which a File is Located (Continued)**RegLocator Table Entries**

Column Name	Value
Signature_	Sig3
Root	2
Key	SOFTWARE\Microsoft\Windows\CurrentVersion\App Paths\isdev.exe
Name	Path
Type	0

GETTING A FOLDER FROM AN INI FILE

In one of the examples you ran when looking at the manipulation of initialization files, you created an initialization file named `DevArtSettings.ini` in the Windows folder. Since the component to which this operation is associated has no component ID, the initialization file was not removed when you uninstalled the Developer Art application. Now you can use this fact to search for a value in an initialization file.

The first thing that you need to do is open this initialization file and change the value for the `InstallLocation` keyword to something that exists on the system. For this example, you can use the installation location for InstallShield Developer. Modify the `DevArtSettings.ini` file accordingly. You now have to add one row each in the `AppSearch` and the `IniLocator` tables. The values that you need to use are shown in Table 10-21.

Table 10-21: Searching for a folder identified in an .ini file.**AppSearch Table Entries**

Column Name	Value
Property	SEARCH
Signature_	Sig1

IniLocator Table Entries

Column Name	Value
Signature_	Sig1
Filename	DevArtSettings.ini
Section	BasicSettings
Key	InstallLocation
Field	0
Type	0

Essentially, this example verifies that the location retrieved from the initialization file exists on the target system. If it exists, the property is set to this value and if it does not exist, the property is not set. This search has nothing at all to do with whether the value exists in the initialization file itself.

Retrieving Raw Values

Here you will follow an example that retrieves the version of Internet Explorer that is installed on the target machine. You do this by retrieving this value from the registry. When you retrieve a raw value all you are getting is the value in either the registry or the initialization file. As long as there is a value to retrieve, the property in the AppSearch table is set to this value. This is different from retrieving a file or a folder

location from the registry or an initialization. In this case, the file or folder has to exist on the target machine before the property is set.

To run this example, you need to create one row in each of the AppSearch and the RegLocator tables. The entries in these two tables are shown in Table 10-22.

Table 10-22: Searching for a raw registry value.

AppSearch Table Entries

Column Name	Value
Property	SEARCH
Signature_	Sig1

RegLocator Table Entries

Column Name	Value
Signature_	Sig1
Root	2
Key	SOFTWARE\Microsoft\Internet Explorer
Name	Version
Type	2

This example looks like the one you did to extract a location from an initialization file. The only main difference is that you used a value of 2 in the Type column to specify that you want to get the value of the registry value name provided in the name column.

Checking for Compliance

Compliance checking uses the same search mechanism to determine if a user is in compliance with an application's installation requirements. To implement compliance checking, the AppSearch table is not required. You will use the Signature, CompLocator, RegLocator, IniLocator, and DrLocator tables. In place of the AppSearch table, you use the CCPSearch table. The acronym CCP stands for Compliance Checking Program.

The CCPSearch table is a table with one column. In this column you place the signatures for the files, folders, and/or values for which you want to search. You can place as many signatures in this table as are required to verify that the user is in compliance. As soon as one of the signatures is found to exist on the target system, the compliance checking terminates and the property CCP_Success is set to 1. The fact that there is a built-in property that gets set when the search is successful is the reason that there is no Property column in the CCPSearch table.

The CCPSearch and the RMCCPSearch actions read the CCPSearch table and perform the necessary compliance check. The CCPSearch action checks the fixed drives on the target system. If this action is unsuccessful in verifying compliance, the RMCCPSearch action can be used to check for any of the signatures listed in the CCPSearch table on removable media. The RMCCPSearch action will check the media in the drive that is defined by the CCP_DRIVE property. This property normally has to be set from the user interface or from a custom action.

The final mechanism is to use the CCP_Success property in a condition that if it is set to 1 will allow the installation to proceed. If the CCP_Success property is not set then the installation is terminated as not being in compliance with the requirements for installing the software.

Using Event Handlers for Searching

There are two event handlers that are called during the fresh install of a Standard project named OnCCPSearch and OnAppSearch. As discussed in Chapter 4 these functions are called in the following order:

```
OnCCPSearch();  
OnAppSearch();
```

```
OnFirstUIBefore();
```

These event handlers are not called during a maintenance installation or any other type of installation. The default implementation for the `OnCCPSearch` and `OnAppSearch` event handlers is a no-op. You could create your own functions and call these functions from inside the `OnFirstUIBefore` or the `OnMaintUIBefore` event handlers.

`InstallScript` provides a number of built-in functions that can be used to perform searches of the target system. The following sections look at implementing some of the same types of searches you performed using the database tables. It was stated earlier that it is best to use the Windows Installer approach to searching; however you can use legacy code that is already available and working.

What you do in this section is applicable only to Standard projects. You will use the same Standard project you have using in this chapter. You will also use the `SearchFeedback` function you created to indicate if the search is successful.

Searching for a File

`InstallScript` provides the `FindFile` and the `FindAllFiles` functions that can be used to search for a file. It also provides a number of functions for accessing folders and drives on the target system.

SEARCHING ALL FIXED DRIVES

Searching for a file on all fixed drives can be accomplished as shown in Figure 10-39. You would do this if you had no idea where the file may be located.

```

////////////////////////////////////
//
//  FUNCTION:   OnAppSearch
//
//  EVENT:     The OnAppSearch event is called prior to calling the
//             OnFirstUIBefore event.
//
////////////////////////////////////
function OnAppSearch()
STRING  szDir, szFileName, svResult;

```

Figure 10-39: *Using InstallScript to search for a file on all fixed drives.*

```

INT     iReturn, iListReturn;
LIST    FixedDrives;
begin

    // Define the file for which to search.
    szFileName = "isdev.exe";

    // Create a list that will hold all
    // fixed drives on the target system.
    FixedDrives = ListCreate(STRINGLIST);

    // Get the list of fixed drives on the target machine.
    GetValidDrivesList(FixedDrives, FIXED_DRIVE, -1);

    // Set the list index to the first item in the list.
    iListReturn = ListSetIndex(FixedDrives, LISTFIRST);

    // Loop through all items in the list of fixed drives.
    while(iListReturn != END_OF_LIST)
        // Get the name of the current fixed drive.
        ListCurrentString(FixedDrives, szDir);

        // Concatenate the root directory specifier
        // to the drive letter.
        szDir = szDir + ":\\";

        // Perform a search for the current fixed drive in the list.
        iReturn = FindAllFiles(szDir, szFileName, svResult, RESET);

        // If the search is successful then set the SEARCH
        // property to the absolute path of the file
        // and then terminate the search process.
        if(iReturn = 0) then
            Msi SetProperty(ISMSI_HANDLE, "SEARCH", svResult);
            // Jump to the end after the property has been set.
            goto Finish;
        endif;

        // As long as the search has not been successful
        // increment the list index to point at the next
        // fixed drive letter.
        iListReturn = ListSetIndex(FixedDrives, LISTNEXT);

    endwhile;

Finish:
;
end;

```

Figure 10-39: *Continued.*

The code shown in Figure 10-38 performs the same search as you performed using the AppSearch action and the related tables in the database. The code first obtains a list that contains all the fixed drives on the target system. It then loops through this list and searches for the defined file. As soon as the search is successful, the code exits the loop after setting the SEARCH property to the search results.

Though this is exactly the same functionality produced with the Windows Installer, you are better off not using InstallScript unless you already have a significant investment already made in legacy code. The main benefit of using the built-in functionality provided by the Windows Installer is that the search that it provides is significantly faster than the search implemented in InstallScript code.

SPECIFYING THE SEARCH PATH FOR A FILE

This example narrows the search path by looking in the registry for the search's starting point. It also limits the search to this one location instead of searching all of the child folders as well. The code for this search process is shown in Figure 10-40.

```

////////////////////////////////////
//
//  FUNCTION:   OnAppSearch
//
//  EVENT:     The OnAppSearch event is called prior to calling the
//             OnFirstUIBefore event.
//
////////////////////////////////////
function OnAppSearch()
STRING  svResult, szFileName, svValue, szName, szKey;
INT     iReturn, nvSize, nvType;
begin

    // Define the file for which to search.
    szFileName = "isdev.exe";

    // Define the key for which you want the value.
    szKey = "SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\" +
           "App Paths\\isdev.exe";

    // Define the value name for
    // which you want the data.
    szName = "Path";
    // Set the root key under which you will work.
    RegDBSetDefaultRoot(HKEY_LOCAL_MACHINE);

```

Figure 10-40: *Searching a specific location for the existence of a file.*

```

// Get the value data of the value name you want to use
// as the starting point of our search.
RegDBGetKeyValueEx(szKey, szName, nvType, svValue, nvSize);

// Perform a search for the file in the one location.
iReturn = FindFile(svValue, szFileName, svResult);

// If the search is successful then set the SEARCH
// property to the absolute path of the file.
if(iReturn = 0) then
    svResult = svValue ^ svResult;
    MsiSetProperty(ISMSI_HANDLE, "SEARCH", svResult);
endif;

end;

```

Figure 10-40: *Continued.*

To follow this example, type the code into the Script Editor or copy it from the CD-ROM at the back of the book. The example performs a search for a file in a specific location using the `FindFile` InstallScript function. It first uses the InstallScript registry functions to obtain the search location. This location is defined by the Path registry value name just as when you used the `RegLocator` table. If the file is found, you need to concatenate the file name and the search path before setting the value of the `SEARCH` property. If you wanted to search all folders below the starting point, you would have had to use the `FindAllFiles` function.

Windows Installer simplifies the creation of a file signature that includes such things as version, size, and creation date. To implement a check against more than just the file name, you can use InstallScript, but the coding is more complicated. You need to make multiple calls to the `GetFileInfo` InstallScript function and compare the results against a predetermined list of values that you could keep in an array.

Searching for a Folder

In this section you will perform the same operations as you did when using the Windows Installer functionality. First, you will obtain the folder in which a certain file exists and then you will obtain a path from an initialization file and search to make sure that it exists on the target system.

GETTING THE FOLDER IN WHICH A FILE EXISTS

This example searches for the file ATL.MSM and then gets the folder in which this file is located (Figure 10-41).

```

////////////////////////////////////
//
// FUNCTION:   OnAppSearch
//
// EVENT:      The OnAppSearch event is called prior to calling the
//              OnFirstUIBefore event.
//
////////////////////////////////////
function OnAppSearch()
STRING  svResult, szFileName, svValue, szName, szKey, svReturnString;
INT     iReturn, nvSize, nvType;
begin

    // Define the file for which to search.
    szFileName = "ATL.MSM";

    // Define the key for which you want the value.
    szKey = "SOFTWARE\Microsoft\Windows\CurrentVersion\\" +
           "App Paths\isdev.exe";

    // Define the value name for
    // which you want the data.
    szName = "Path";

    // Set the root key under which you will work.
    RegDBSetDefaultRoot(HKEY_LOCAL_MACHINE);

    // Get the value data of the value name you want to use
    // as the starting point of our search.
    RegDBGetKeyValueEx(szKey, szName, nvType, svValue, nvSize);

    // Perform a search for the file BELOW the specified location.
    iReturn = FindAllFiles(svValue, szFileName, svResult, RESET);

    // If the file search is successful then set the SEARCH
    // property to the absolute path of the file but not
    // including the file name.
    if(iReturn = 0) then
        ParsePath(svReturnString, svResult, PATH);
        MsiSetProperty(ISMSI_HANDLE, "SEARCH", svReturnString);
    endif;

end;

```

Figure 10-41: *Getting the folder in which a file is located.*

This example is a simple extension of the last example where you searched for a file in a specific location. Here you use the FindAllFiles function and then strip off the file name before you set the value of the SEARCH property.

GETTING A FOLDER FROM AN INI FILE

This example uses an InstallScript function to obtain the value of InstallLocation keyword in the DevArtSettings.ini file and then checks to see if this folder exists on the target system. If it exists, the code then sets the value of the SEARCH property to this value. The code for this is shown in Figure 10-42.

```

////////////////////////////////////
//
//  FUNCTION:    OnAppSearch
//
//  EVENT:      The OnAppSearch event is called prior to calling the
//              OnFirstUIBefore event.
//
////////////////////////////////////
function OnAppSearch()
STRING  svResult, szFileName, szSectionName, szKeyName;
INT     iReturn;
begin

    // Set the parameters for accessing the INI file.
    szFileName = "DevArtSettings.ini";
    szSectionName = "BasicSettings";
    szKeyName = "InstallLocation";

    // Get the value of the InstallLocation keyword.
    GetProfString(szFileName, szSectionName, szKeyName, svResult);

    // Verify that the folder exists.
    iReturn = ExistsDir(svResult);

    // If the folder exists then set the SEARCH
    // property to the value obtained from the INI file.
    if(iReturn = 0) then
        Msi SetProperty(ISMSI_HANDLE, "SEARCH", svResult);
    endif;

end;

```

Figure 10-42: *Verifying that a folder in an initialization file exists on the target system.*

In this example you obtain the value of the `InstallLocation` keyword from the initialization file and then check for its existence. You do not need to pass the absolute path of the initialization file to the `GetProfString` function since the `DevArtSettings.ini` file is located in the `Windows` folder.

Retrieving Raw Values

The final example in this section demonstrates how to retrieve the value from the registry and, if this value exists, how to set the `SEARCH` property to this value. The code for this operation is shown in Figure 10-43.

```

////////////////////////////////////
//
// FUNCTION:   OnAppSearch
//
// EVENT:     The OnAppSearch event is called prior to calling the
//            OnFirstUIBefore event.
//
////////////////////////////////////
function OnAppSearch()
STRING  svValue, szName, szKey;
INT     iReturn, nvSize, nvType;
begin

    // Set the registry parameters.
    szKey = "SOFTWARE\\Microsoft\\Internet Explorer";
    szName = "Version";

    // Set the root registry key to be used.
    RegDBSetDefaultRoot(HKEY_LOCAL_MACHINE);

    // Get the value data for the Version value name.
    iReturn = RegDBGetKeyValueEx(szKey, szName, nvType,
                                svValue, nvSize);

    // If the registry value exists then set the SEARCH
    // property to this value.
    if(iReturn = 0) then
        Msi SetProperty(ISMSI_HANDLE, "SEARCH", svValue);
    endif;

end;

```

Figure 10-43: *Obtaining the raw value form the registry and if it exists setting a property.*

We have now completed the discussion on searching the target system during an installation. Even though you have spent some time performing searching using InstallScript code, it is recommended that you use the Windows Installer functionality unless there is something that cannot be accomplished with it.

Miscellaneous Operations

Before you end this chapter you need to take a brief look at two minor operations that you will have need of at one time or another. The first operation is the creation of a launch condition and the second operation is how to create empty folders during an installation.

Specifying Launch Conditions

A launch condition is a logical statement that can prevent the end user from installing your application if the system attributes are not adequate to properly run the application. Though you could create a check for the operating system attributes in the OnBegin event handler, it is best to use the functionality provided by the Windows Installer.

The Windows Installer, using the LaunchConditions standard action and the LaunchCondition table, implements launch conditions. The LaunchCondition action reads the conditions in the LaunchConditions table and terminates the installation if any of the conditions returns FALSE.

The LaunchConditions table has two columns. The first column contains the condition and the second column contains a message that is displayed if the condition is FALSE. Both columns require values. To populate the LaunchConditions table, use the Product Properties sub-view in the General Information view. The Install Condition property in the Product Properties sub-view allows you to define your launch conditions.

Click the ellipsis in the Install Condition property field to launch the Product Condition Builder dialog (Figure 10-44).

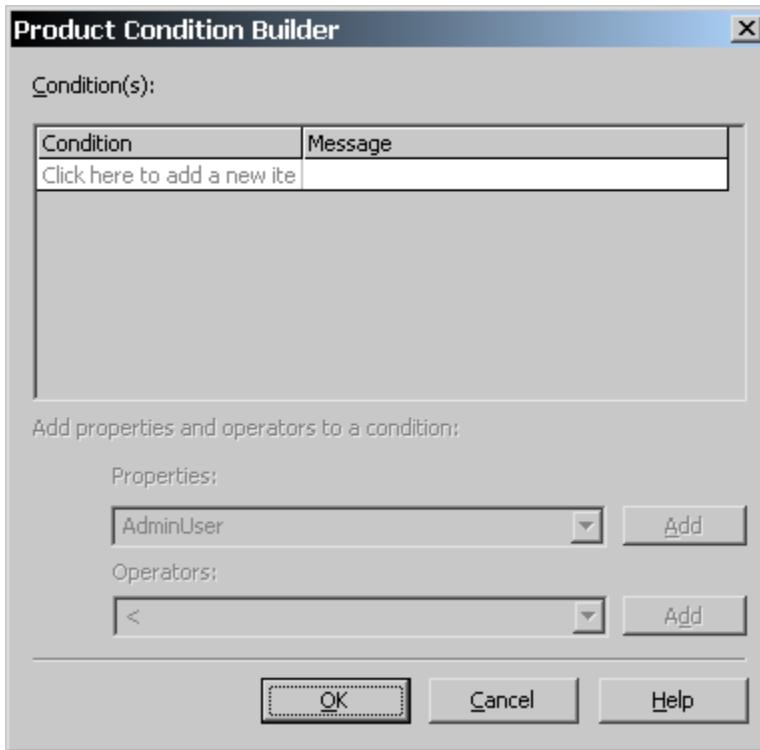


Figure 10-44: *The Product Condition Builder dialog box for creating launch conditions.*

Click in the Condition field of the dialog to enable the drop-down menus at the bottom of the dialog. You can use these menus to select the properties and operators with which to create a condition statement. You can combine condition statements with logical operators such as AND and OR.

As an example, you can create a launch condition that checks to see if the target operating system is Windows 2000 with service pack 1 installed. To do this, you can combine these two conditions using the AND logical operator, or place them in separate rows to have specific messages for each part of the condition. For this example, the two conditions that you enter in the Product Condition Builder dialog are shown in Table 10-23.

Table 10-23: Example of Install Conditions

Condition	Message
VersionNT = 500	This application requires that Windows 2000 be installed before installing this application.
ServicePackLevel = 1	This application cannot run unless service pack 1 is installed.

Both VersionNT and ServicePackLevel are properties that the Windows Installer sets when an installation package is launched. The first condition checks to see if the operating system is version 5.0. Version 5.0 is the version of Windows 2000. The Windows Installer help contains an appendix topic that provides all the valid values for operating system properties, *Operating System Property Values*. The second condition checks if service pack 1 is installed. There is a separate message for each of these conditions so that the user can receive more detailed feedback about what may be wrong if the installation terminates.

Creating Empty Folders

You might want your installation to create an empty folder so the application has a default location for creating application-generated files. The best approach to creating an empty folder is to have the main component in the application create this empty folder. Because components are the basic installation units, you need to associate the action of creating an empty folder with a component. You can do this by using the CreateFolder table and the Directory table in the Direct Editor. The CreateFolder table has two columns that tie a location defined in the Directory table to a component in the Component table.

As a simple example, you can see how to create a folder named "DevArt Data" and have this folder created under the Application Data folder defined by the operating system. The first entry you need to make is in the Directory table. The Directory table entry is shown in Figure 10-45.

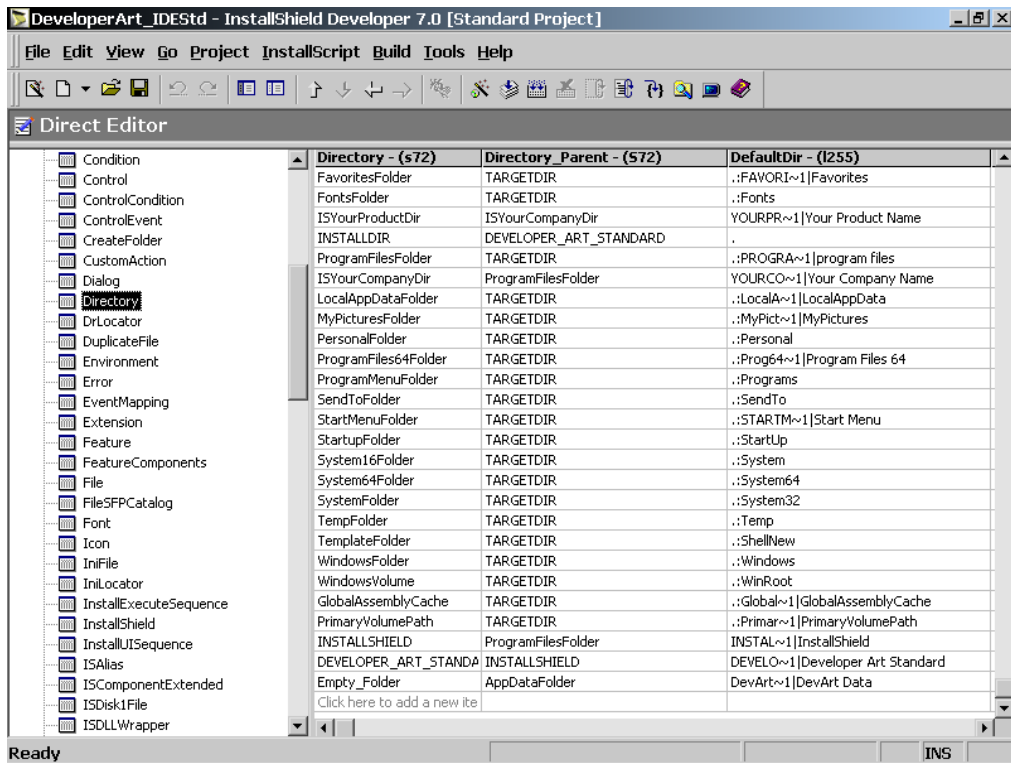


Figure 10-45: The Directory table in the Direct Editor with the Empty_Folder entry.

The Directory table shown in Figure 10-45 contains an entry that defines the location specified by the Empty_Folder identifier. The Directory column entry is the name of the identifier that you have created for this particular location. The Directory_Parent column contains the identifier for the path to the Application Data folder that is defined by the operating system. When you click in this second column, a drop-down menu lists all the identifiers in the Directory table. Select the identifier that you want to use. Finally, in the DefaultDir column, enter the name of the empty folder that you want created under the Application Data folder. Since the folder you want to create has a space in the name and is longer than eight characters you need to place both the full name and the short name of this folder in to the third column. Separate the two naming conventions with a vertical bar.

The next thing that you have to do is tie the entry in the Directory table to the DeveloperArt component using the CreateFolder table. In the CreateFolder table, the

entries for both columns are available from drop-down menus. The entry required for this example is shown in Figure 10-46.

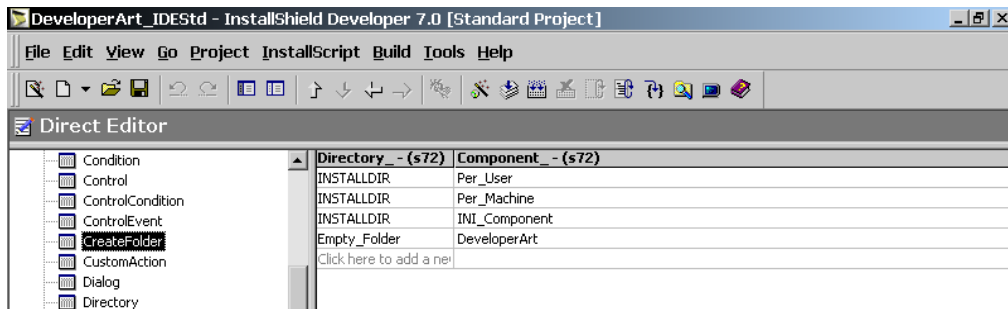


Figure 10-46: *The CreateFolder table in the Direct Editor to connect the Developer.Art component to the Directory table entry.*

To test this example, build your project and install the Developer Art application. You will see that an empty folder named "DevArt Data" is created.

An easier method for creating an empty folder is to create a special component that has as its only function the creation of this folder. This can be done in the Files view under Step 2. For every folder that you create in this view InstallShield Developer creates a component that, when installed, will create that folder.

Conclusion

This chapter discusses a number of the tasks that need to be carried out when an installation package is created. You have seen that there are many tables that write to the registry. However, you have also seen you need to use the Registry table to create application-related entries.

InstallShield Developer's many views make it easier for you to define the operations that your installation package needs to perform. There are views that permit you to create file associations and MIME types, make entries in initialization files, and to create environment variables. However, there is no built-in functionality in Windows Installer to read environment variables during an installation. To do this, you need to write code and to place that code in a custom action or in the user interface script used by a Standard project.

You learned that there is a very robust search mechanism implemented by the Windows Installer and this mechanism is fairly fast even when performing a search for a file on a fixed drive. The search mechanism can be used to search for files, folders, initialization file entries, and registry entries. As part of this searching process, you can also retrieve raw values from initialization files and from the registry.

This chapter also discusses how you could use InstallScript to implement searching. There are a number of InstallScript functions that are very useful in this regard. However, searching using InstallScript functions is significantly slower than using the built-in Windows Installer functionality. It is recommended that you use the Windows Installer to perform all searches, except in those special cases where InstallScript can perform the task more efficiently.

Finally, this chapter looked at how to create conditions that can be used to analyze the operating system at the beginning of an installation to make ensure that all the requirements are present for the application to run correctly. Also, in the last section, learned how to define a folder that is created during the installation.

Chapter

11

InstallScript Custom Actions

To this point, all example InstallScript programs have been inside an explicit `program...endprogram` block. This works for Standard projects where you want to perform operations in the user interface sequence. However, if you want to use InstallScript in the execute sequence of a Standard project or in both the user interface sequence and execute sequence of a Basic MSI project, you need to create InstallScript functionality inside of a custom action. If you want to make changes to the target system using InstallScript, you need to use deferred custom actions.

This chapter introduces the techniques that you need in order to create InstallScript custom actions. The techniques discussed here are used in several locations later in the book to perform installation operations. The material discussed in Chapters 3 and 4 provides a foundation for what is covered in this chapter.

Creating the Project

The first thing that you need to do is create a new project to practice with custom actions. The Standard project that you have been using is not adequate for this purpose. In this case you will create a Basic MSI project so you can use custom actions in both the user interface and the execute sequence tables.

As with the Standard project that was used in the previous chapters on InstallScript, you do not want to wade through a bunch of user interface dialogs every time that you run an example. Also, since this time you will be running the execute sequence table actions as well as the user interface sequence actions, you do not want to contend with having to perform an uninstallation every time you run an example. Accordingly, you will make a few changes to the Basic MSI project after creating it so you can use it to practice with InstallScript custom actions.

When you create this Basic MSI project give it the name of "Learning ISScript Custom Actions". This is the name of the project on the CD-ROM at the back of the book. After you have created this project start with the General Information views under Step 1 and enter the requested information. Then you will need to customize this project to make it appropriate for working with custom actions but not having to contend with a user interface or having to uninstall every time you run a test.

Preventing the User Interface Dialogs From Running

Click the Sequences view in Step 5 where you can make some very simple changes to avoid having a user interface displayed. For this project, you want to prevent the user interface dialogs from running. A good way to do this is to modify the condition for these dialogs so that the condition evaluates to FALSE and thus prevents the dialogs from being displayed. An easy way to modify the conditions on these dialogs is to use a non-existent property. When the condition is evaluated, it evaluates to FALSE. You can use a public property such as `CONDITION` as this non-existent property. For the `SetupInitialization` and the `SetupProgress` dialogs, this property will constitute the complete condition. For the `SetupInitialization` this condition will look like what is shown in Figure 11-1.

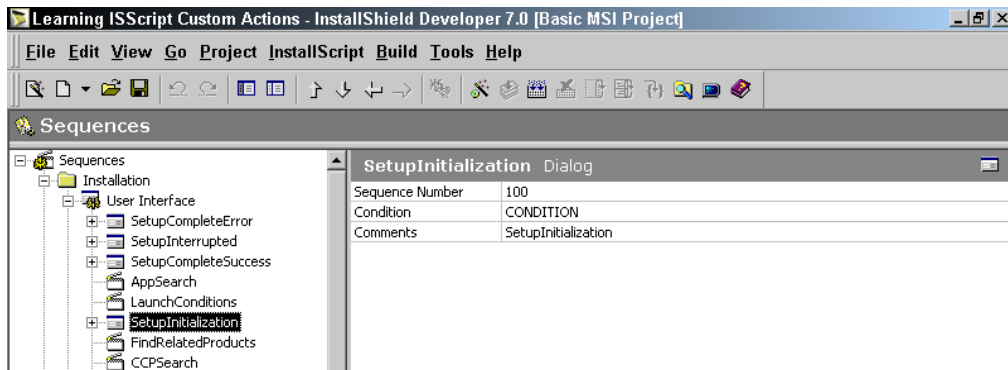


Figure 11-1: *Conditioning the SetupInitialization dialog so that it is not displayed.*

The condition placed on the SetupProgress dialog will be exactly the same as shown for the SetupInitialization dialog as shown in Figure 11-1. The condition placed on the InstallWelcome dialog is a modification of the default condition. You need to AND the CONDITION property to the condition that is already defined. The result of this is shown in Figure 11-2.

You do not need to condition the final dialogs that are displayed when an installation is complete or has been terminated for some reason. This is so there is some visual evidence that the process is completed successfully or that there was an error. These final dialogs are the ones at the top of the user interface sequence and they have the negative sequence numbers of -1, -2, and -3.

You do not need to worry about any of the other dialogs that appear in the User Interface sequence because, after this project is installed, the Windows Installer will not recognize that anything has been installed. Because of this, the only user interface wizard that would ever run is the fresh install wizard that begins with the InstallWelcome dialog. Now anytime that you want to run the installation package so it shows the user interface, all you have to do is give a value to the CONDITION property. You do this in the Property Manager under Advanced Views. You can, of course, set the value of the CONDITION property on the command line as well if and not have to rebuild the project. As discussed in Chapter 4 you can insert the command line that you want passed to the Windows Installer in Setup.ini.

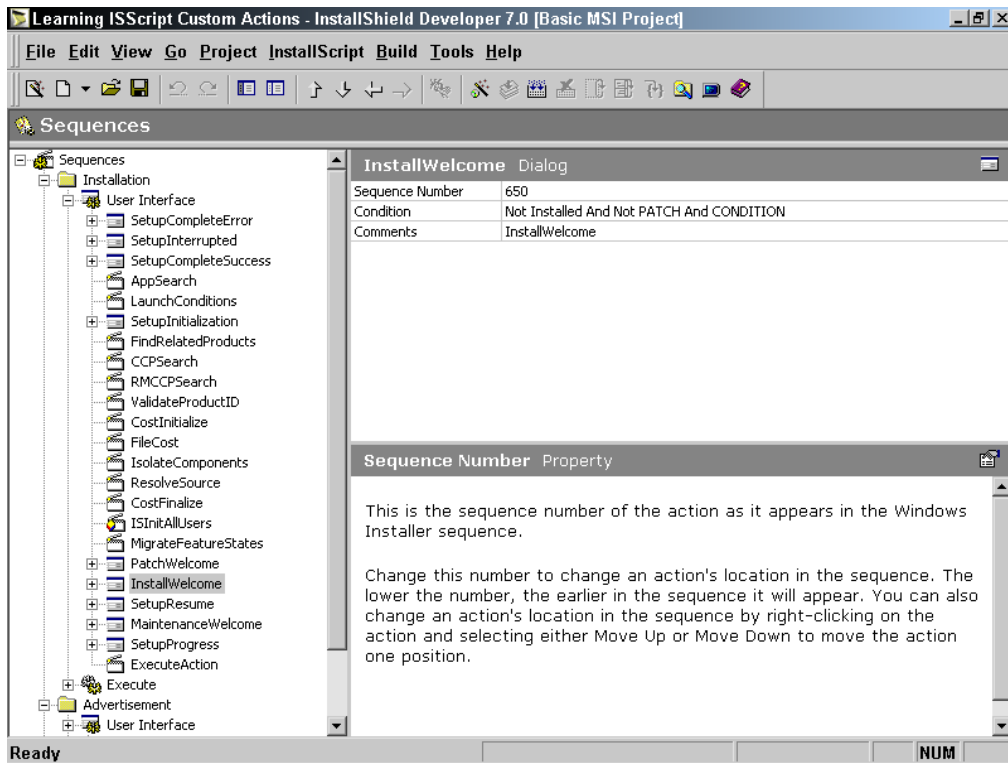


Figure 11-2: *The condition on the InstallWelcome dialog.*

After you build the project, as a final touch to the user interface modifications, you can eliminate the initialization dialog launched by Setup.exe by adding the keyword/value entry `UI=0` to the end of the `[Startup]` section in the `Setup.ini` file just as you did for the "Learning InstallScript" project. This was explained in detail in Chapter 6.

Preventing Project Registration

Next, you want to prevent the Windows Installer from registering this example project every time that you test your custom actions. To do this, use the same `CONDITION` property to prevent certain actions from being executed. The actions that you want to keep from running are the `RegisterUser`, `RegisterProduct`, `PublishFeatures`, and `PublishProduct` actions. These actions are in the `Execute` sequence and the condition on the `RegisterUser` action should look like what is

shown in Figure 11-3. The condition for the other three actions will be exactly the same as shown in Figure 11-3.

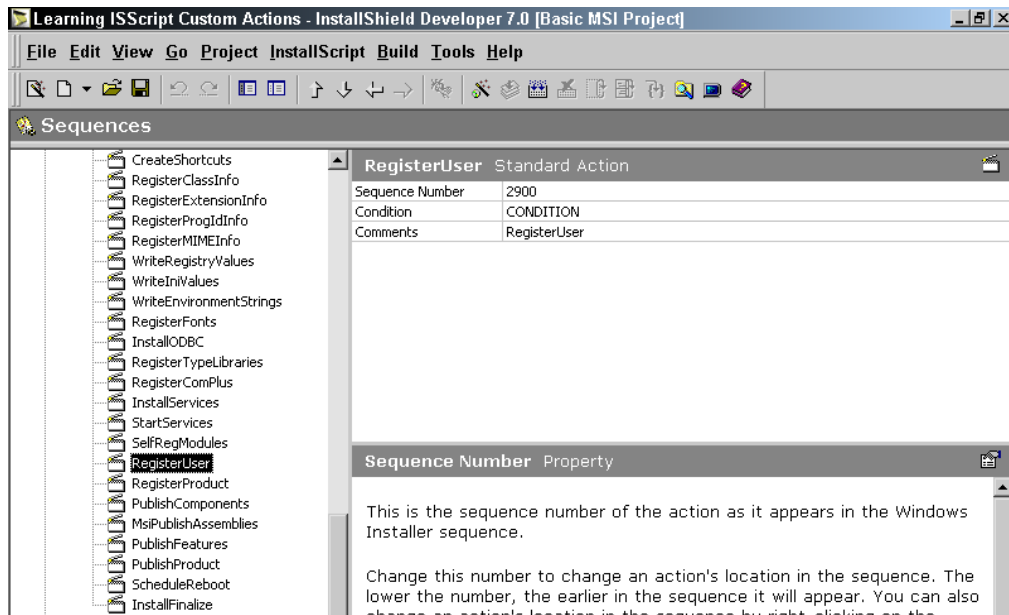


Figure 11-3: The condition on the RegisterUser action in the execute sequence.

Defining a Feature and Component

You have to define one feature and one component for this project or a Windows Installer error occurs when the installation is run. You do not want to leave any artifacts on the target machine with regard to the registration of a component even if it does not have a file in it. Although you could do what you did earlier and place a condition on the ProcessComponents action in the Execute sequence, there is a slightly more instructive way to ensure this. You need to delete the Component Code property of the component so it is NULL. A component that does not have a component ID is not registered on the target system.

To create one feature and one component, do the following:

1. Go to Advanced Views and click on the Setup Design view.

2. Right click on the Setup Design node and select the New Feature option. Name this feature Feature.
3. Right click on the feature and select the New Component option. Name this component Component.
4. Delete the GUID provided for the Component's Component Code property. This should look like what is shown in Figure 11-4.

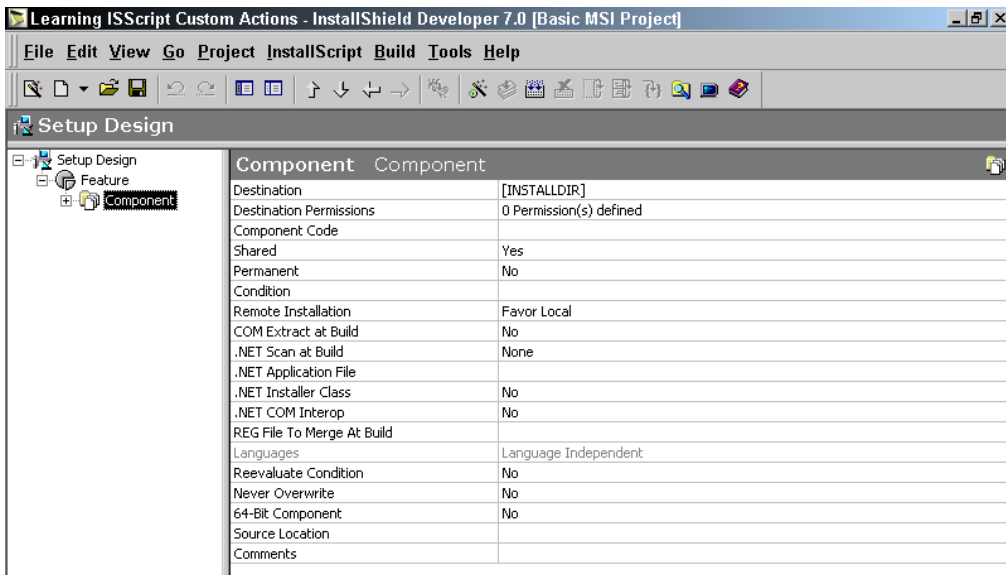


Figure 11-4: *The Feature/Component tree in the Learning ISScript Custom Actions project.*

Because you are not including a file in this component, you can now run the custom action examples that you are going to create and not leave any trace of this project on your machine. You can run each example without having to uninstall between the examples.

Building the Project

The last thing you need to do is to build this project. Do this by clicking the Build button on the toolbar. After the build completes, a warning appears in the Output Window stating that there are no files in the project. This warning will not appear

again because you will be reinserting your compiled and linked scripts into the Binary table just as you did in the previous chapters on InstallScript. After the build is complete remember to make the entry in the Setup.ini file as you did in Chapter 6.

Now the project will show almost no user interface. The project will display the finish dialogs and another dialog that appears on the screen for a short time. This dialog is the built-in initialization dialog that is displayed by the Windows Installer engine. You could eliminate this dialog, but that requires you to run the installation silently. However, running the installation in silent mode does not run any custom action that is placed in the user interface sequence.

Now that the project setup is complete, you can create your first InstallScript custom action.

Creating an InstallScript Custom Action

To create a custom action using InstallScript, follow a three-step process.

1. Create the InstallScript function that will be executed as the target of the custom action. You will not be able to compile this InstallScript code until you complete Step 2.
2. Create a custom action that targets the exported function in the InstallScript code. Compile the InstallScript code.
3. Insert the custom action into a sequence table or attach it to a button on a dialog box.

The implementation details of these steps are covered in the following three sections. By the time you are finished you will have worked through the Custom Action Wizard to create an InstallScript custom action.

Creating the InstallScript Function

The first operation is to create an InstallScript function that will be the target of a custom action. To create the initial template for this function, do the following:

1. Go to the InstallScript view under Step 5.
2. Right click on the Files icon and select the New Script File option from the context menu (Figure 11-5).
3. Click on the Setup.rul file in order to display the contents of the template in the Script Editor.

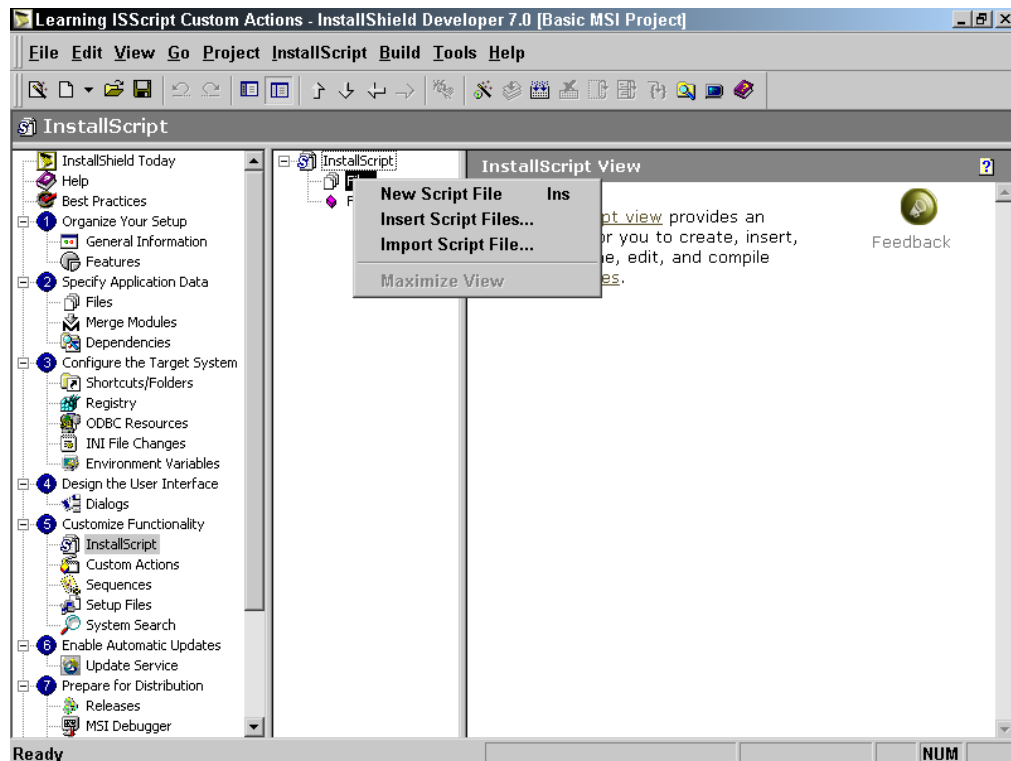


Figure 11-5: Creating a new Setup.rul file for use with custom actions.

The name of the new script file is Setup.rul and it contains a template for creating a custom action. The template file, as created by InstallShield Developer, is shown in Figure 11-6.

```

////////////////////////////////////
//
//   I I I I I I I   S S S S S S
//   II    SS                      InstallShield (R)
//   II    S S S S S S              (c) 1996-2001,
//   II    SS                      InstallShield Software Corporation
//   I I I I I I I   S S S S S S              All rights reserved.
//
//
// This template script provides the code necessary to build an
// entry-point function to be called in an InstallScript
// custom action.
//
//   File Name:  Setup.rul
//
// Description:  InstallShield script
//
////////////////////////////////////

// Include Isrt.h for built-in InstallScript function prototypes.
#include "isrt.h"

// Include Iswi.h for Windows Installer API function prototypes and
// constants, and to declare code for the OnBegin and OnEnd events.
#include "iswi.h"

// The keyword export identifies MyFunction as an entry-point
// function. The argument it accepts must be a handle to the
// Installer database.
export prototype MyFunction(HWND);

// To Do:  Declare global variables, define constants, and
//         prototype user-defined and DLL functions here.
// To Do:  Create a custom action for this entry-point function:
// 1. Right-click on "Custom Actions" in the Sequences/Actions view.
// 2. Select "Custom Action Wizard" from the context menu.
// 3. Proceed through the wizard and give the custom action a
//    unique name.
// 4. Select "Run InstallScript code" for the custom action type,
//    and in the next panel select "MyFunction" (or the new name of
//    the entry-point function) for the source.
// 5. Click Next, accepting the default selections until the wizard
//    creates the custom action.

```

Figure 11-6: *Template Setup.rul for creating InstallScript custom actions.*

```

//
// Once you have made a custom action, you must execute it in your
// setup by inserting it into a sequence or making it the result
// of a dialog's control event.

////////////////////////////////////
//
// Function: MyFunction
//
// Purpose: This function will be called by the script engine when
//           Windows(TM) Installer executes your custom action
//           (see the "To Do," above).
//
////////////////////////////////////
function MyFunction(hMSI)
    // To Do: Declare local variables.
begin

    // To Do: Write script that will be executed when MyFunction
    //         is called.

end;

// To Do: Handle initialization code before the sequence
//         (User Interface or Execute) starts.
//         This will be called only once in an installation.
// function OnBegin
// begin
// end;

// To Do: Write clean-up code when the sequence
//         (User Interface or Execute) ends.
//         This will be called only once in an installation.
// function OnEnd
// begin
// end;

```

Figure 11-6: *Continued.*

At the top of this file, there are included the two header files `isrt.h` and `iswi.h`. These two header files include many other header files and these header files prototype both the built-in InstallScript functions and the Windows Installer functions that can be used in InstallScript custom actions.

The code shown in Figure 11-6 provides the prototype for a function that is called `MyFunction`. The prototype for this function is the format that you need to follow for any function that is to be used as the target of a custom action. As discussed in

Chapter 4, the call to an InstallScript custom action from the Windows Installer passes through ISScriptBridge.dll, which in turn calls the exported function in the compiled script. This exported function can have only one argument passed to it and that argument is the handle to the current Windows Installer session. This handle is required as an argument by many of the Windows Installer functions that are exported by msi.dll. The handle to an Windows Installer session is an integer value.

Not all functions have to be prototyped in this fashion, only those that are to be the target of a custom action. There are many other functions that serve as helper functions. These helper functions are called by the functions that are the targets of custom actions to perform operations such as sorting.

Next in the template file is a skeleton definition for the MyFunction function. Finally there are two functions that are commented out. These functions are the only two event handler functions that are supported for use in a Basic MSI project. These two event handlers are the OnBegin and the OnEnd functions. The default implementation of these two functions is a no-op. In other words, these functions do nothing unless you define something for them to do inside your script. The OnBegin function is called at the beginning and is used to perform any initialization actions required for the installation program. The OnEnd function is used to perform any necessary cleanup. These event handler functions are discussed later in this chapter.

The first custom action you will create displays a message box. As part of creating this custom action you will clean up the template Setup.rul file so that it is more manageable. The operations that you want to perform are listed as follows:

1. Delete the comment statements in the template script file as they pertain to instructions for creating an InstallScript custom action. Figure 11-7 shows what you want to have this file look like.
2. Rename the target function from MyFunction to ISScriptCustomAction and this will be the name of the function used throughout this chapter for all examples.
3. Delete the commented-out event handler function definitions.

4. Inside the `ISScriptCustomAction` function insert a call to the `SprintfBox` function to display a message box. You insert a call to this built-in function just as you did in the previous `InstallScript` chapters.
5. Add the definition of a constant that will be used as the caption for the message box to be displayed.

This creates the basic script file as shown in Figure 11-7. This script is much simpler and it makes an excellent starting point for our investigation of `InstallScript` custom actions. You can make life easier by copying the script from the file on the CD-ROM.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates the creation
//                 of a simple custom action that displays
//                 a message box.
//
////////////////////////////////////

#include "isrt.h"
#include "iswi.h"

#define CAPTION    "Feedback"

export prototype ISScriptCustomAction (HWND);

////////////////////////////////////
// Function:      ISScriptCustomAction
//
// Purpose:       This function is the target of an InstallScript
//                 custom action.
////////////////////////////////////
function ISScriptCustomAction (hMSI)
STRING  szFormat;
begin

    szFormat = "Performing an InstallScript custom action...";

    SprintfBox (INFORMATION, CAPTION, szFormat);

end;

```

Figure 11-7: *The basic custom action script.*

You cannot compile this script until you have created a custom action that uses this function as its target. This is discussed in the next section.

Using the Custom Action Wizard

Now that you have your script in place for implementing a custom action, you need to create the custom action that will target this script function. To launch the Custom Action Wizard custom action, do the following:

1. Go to the Custom Actions view under Step 5.
2. Right click on the Custom Actions node and select the Custom Action Wizard option as shown in Figure 11-8. This launches the Custom Action Wizard.

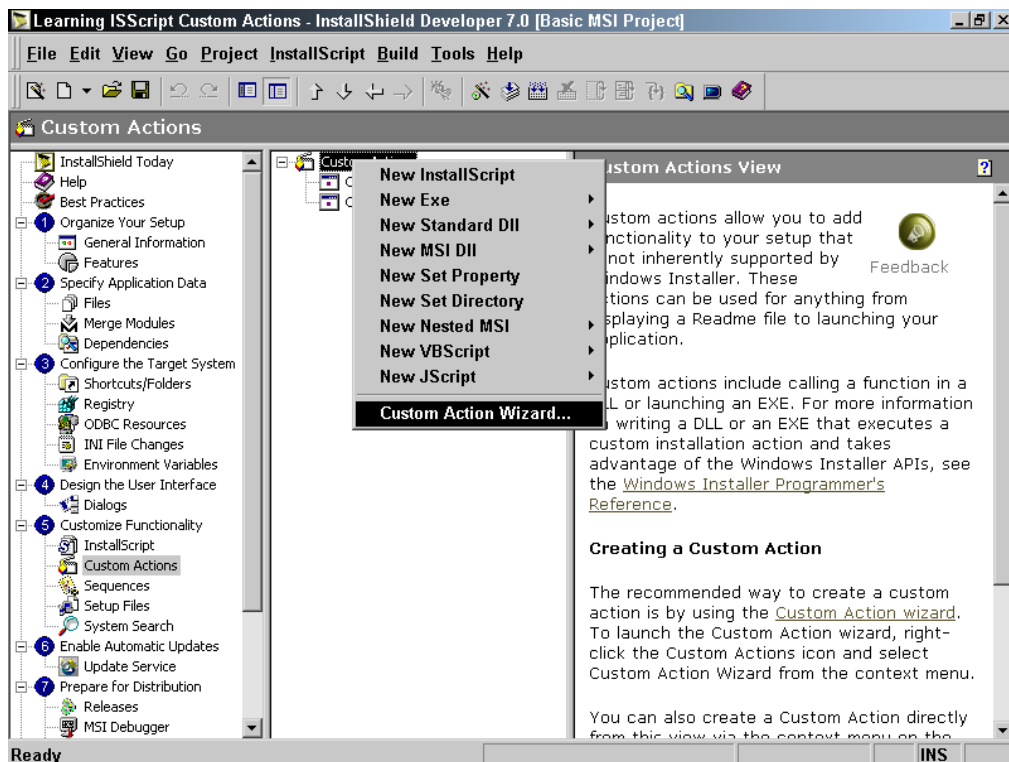


Figure 11-8: The context menu for launching the Custom Action Wizard.

Note that there are nine other options on the context menu shown in Figure 11-8. These other options are used to create custom actions without the use of the Custom Action Wizard. We will discuss these options later in this chapter. Having first used the Custom Action Wizard to create a custom action will make the functionality of these other options much clearer.

When you launch the Custom Action Wizard you will get the wizard Welcome dialog as shown in Figure 11-9.

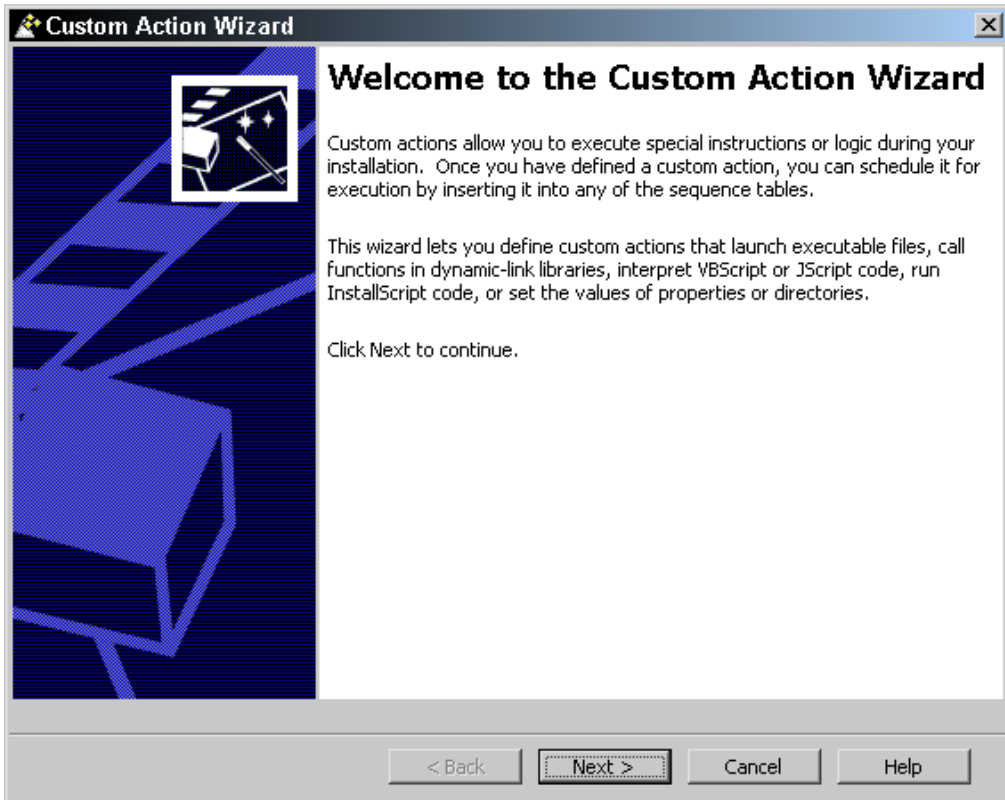


Figure 11-9: *The Welcome dialog of the Custom Action Wizard.*

Since there are no operations to be carried out in the Welcome dialog click Next to move to the Basic Information dialog. In the Basic Information dialog, you need to enter a name for the custom action that you are creating. This name is used to populate the first column of the CustomAction table.

Type `ISScriptCustomAction` in the Name field and optionally type a comment in the Comment field (Figure 11-10). The name of the custom action is how you will insert it into a sequence table. The comment can be used to document the purposes of the custom action and is maintained only in the project file. It is not used in the MSI package.

The screenshot shows a Windows dialog box titled "Custom Action Wizard" with a close button (X) in the top right corner. The dialog is divided into a header section and a main content area. The header section is titled "Basic Information" and contains the instruction "Provide basic information about your custom action" and a small icon of a film strip. The main content area has a grey background and contains the instruction "Enter the name of your custom action and provide a brief description." Below this, there are two text input fields. The first is labeled "Name" and contains the text "ISScriptCustomAction". Below the "Name" field is a note: "The name you give your custom action may contain only letters, numbers, the underscore, or period. The name must begin with a letter or an underscore." The second input field is labeled "Comment" and contains the text "Custom action for learning InstallScript custom actions." Below the "Comment" field is another note: "This comment is stored in your project file for reference and is not used by the Windows Installer at any time." At the bottom of the dialog, there are four buttons: "< Back", "Next >", "Cancel", and "Help".

Figure 11-10: *The entries to be made in the Basic Information panel.*

Click Next to display the Action Type panel (Figure 11-11). Select Run InstallScript Code from the Type drop-down combo box. This is the default option. Chapter 3 discusses the custom actions listed in this combo box. Because you selected Run InstallScript Code, the Location drop-down combo box is disabled. This is because `ISScriptBridge.dll`, which is streamed into the Binary table, manages the running of InstallScript custom actions.

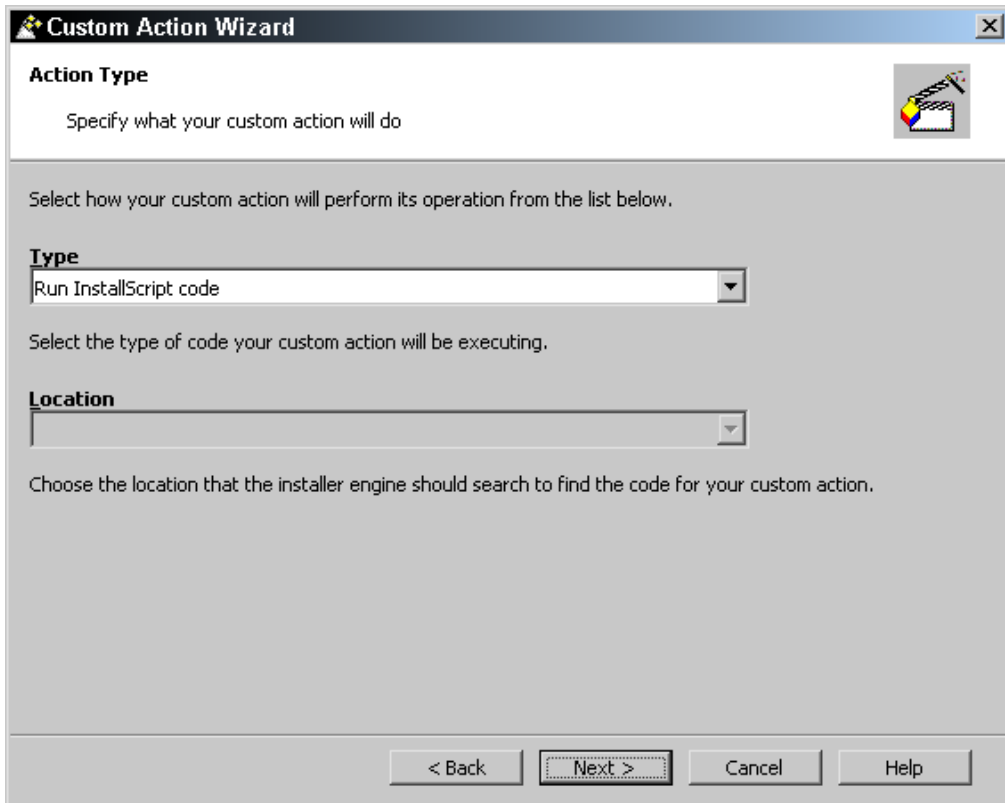


Figure 11-11: *The Action Type panel of the Custom Action Wizard.*

Click Next to move to the Action Parameters panel (Figure 11-12). In the Action Parameters panel, only the Source drop-down menu is enabled. The Source drop-down menu lists all the InstallScript functions that use the export keyword in their prototype. The export keyword tells the wizard which functions are intended to be targets of custom actions. In this example, there is only one function that is prototyped with this keyword, so there is only one function in this menu.

The Source combo box and the Target edit field correspond to two columns in the CustomAction table that have the same names. The Target edit field is disabled because, as discussed in Chapter 4, InstallShield Developer controls the actual name that is used in this column of the CustomAction table. Most likely the name of the function used for this example is `£1`. The `IsConfig.ini` file handles the translation between this name and the name you used in your script.

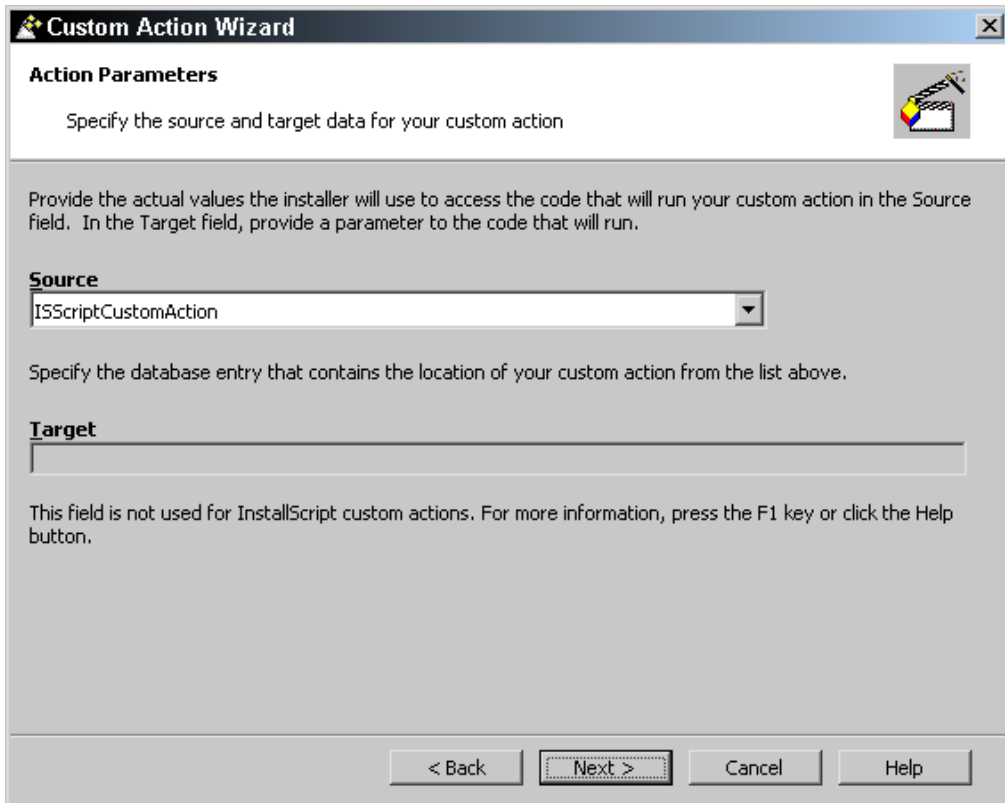


Figure 11-12: *The Action Parameters dialog of the Custom Action Wizard.*

Click Next to move to the Additional Options dialog (Figure 11-13). This is where you make the selections that define your custom action. The first part of this dialog determines how the Windows Installer is to handle return values from the custom action. The first check box is disabled and this indicates that InstallScript custom actions must run synchronously. Other types of custom actions can run asynchronously as well as synchronously. The second check box with the label “Ignore custom action return code” specifies how the Windows Installer should treat the return values from a custom action. If this check box is deselected and a custom action returns a failure return code, Windows Installer terminates the installation. If you select this check box, it does not matter what the custom action returns. The Windows Installer will continue with the installation. For this example, leave this check box deselected.

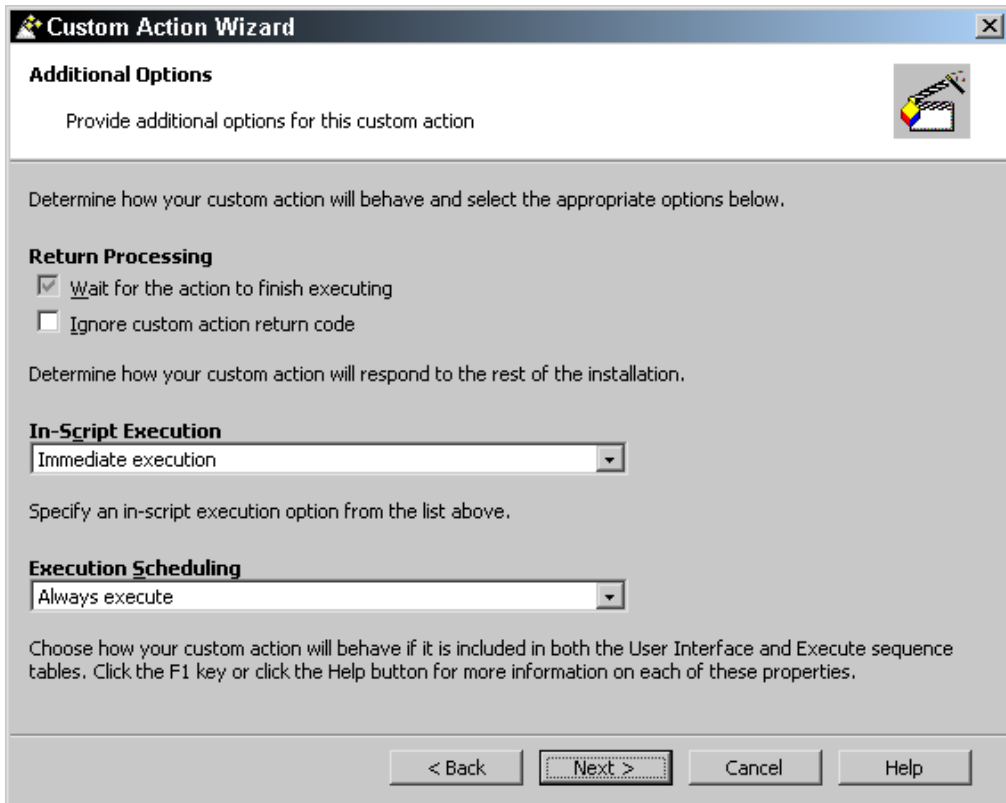


Figure 11-13: *The Additional Options panel of the Custom Action Wizard.*

The value in the In-Script Execution field indicates where the custom action will run. Use the default value, “Immediate execution”. For in-script execution, this option allows the most flexibility in where you can place your custom action. We will look at the use of deferred custom action later in this chapter. All of the options available for this field are described in the following list.

Immediate execution: Windows Installer runs this type of custom action as soon as it is encountered in the Sequence table. This type of custom action can, with some exceptions, be placed at any location in either the user interface or the execute sequence tables. This is also the only type of custom action that can be attached to a button on an authored dialog.

Deferred execution: When Windows Installer encounters this type of custom action, it writes it into the execution script and then, only when the execution script is run will the custom action be executed. This means that a deferred custom action can be placed only in the execute sequence table and only between the InstallInitialize and the InstallFinalize actions.

Rollback execution: This is a deferred execution custom action that runs only during a rollback of the installation. Otherwise, it is handled like the deferred execution custom action.

Commit execution: This is a deferred execution custom action that runs only at the end of a successful installation. Otherwise, it is handled like the deferred execution custom action.

Deferred execution in System context: This is a deferred execution custom action that runs with local system privileges in a managed environment. Normally a custom action will have the same privilege level as the user performing the installation.

The Execution Scheduling section of the Additional Options panel indicates when a custom action will be executed based on the sequence tables into which is inserted or the process used to run the custom action. For this beginning custom action, accept the default value, “Always execute.”

The four options are described in the following list:

Always execute: The custom action may run twice if present in both Sequence tables. On Windows 9x machines, the custom action runs only once if the installation is run with a full or a reduced user interface, even if the custom action is inserted into both Sequence tables. On Windows NT/2000 machines, a custom action runs twice if placed in both sequence tables. As long as the sequence table is run, the custom action runs if it has been inserted into that table. The user interface level used does not impact this.

Execute only once: The custom action executes once if present in both sequence tables. The custom action is not run in the execute sequence if the user interface sequence has run. It does not matter whether the custom action has been inserted into the user interface sequence or not. The operation on Windows NT/2000 and Windows 9x machines is the same.

Execute only once per process: The custom action on a Windows NT/2000 machine operates just the same as if it had been identified as “Always execute.” This has an impact on Windows 9x machines because there is only one process on these machines. On a Windows 9x machine, this custom action runs in the execute sequence table only if the user interface table has not been run.

Always execute at least once on the client: This custom action does not run on a Windows NT/2000 machine. On a Windows 9x machine, this custom action is just the reverse of the “Execute only once per process” type of custom action. Here the custom action runs only in the execute Sequence table if the user interface table has been run.

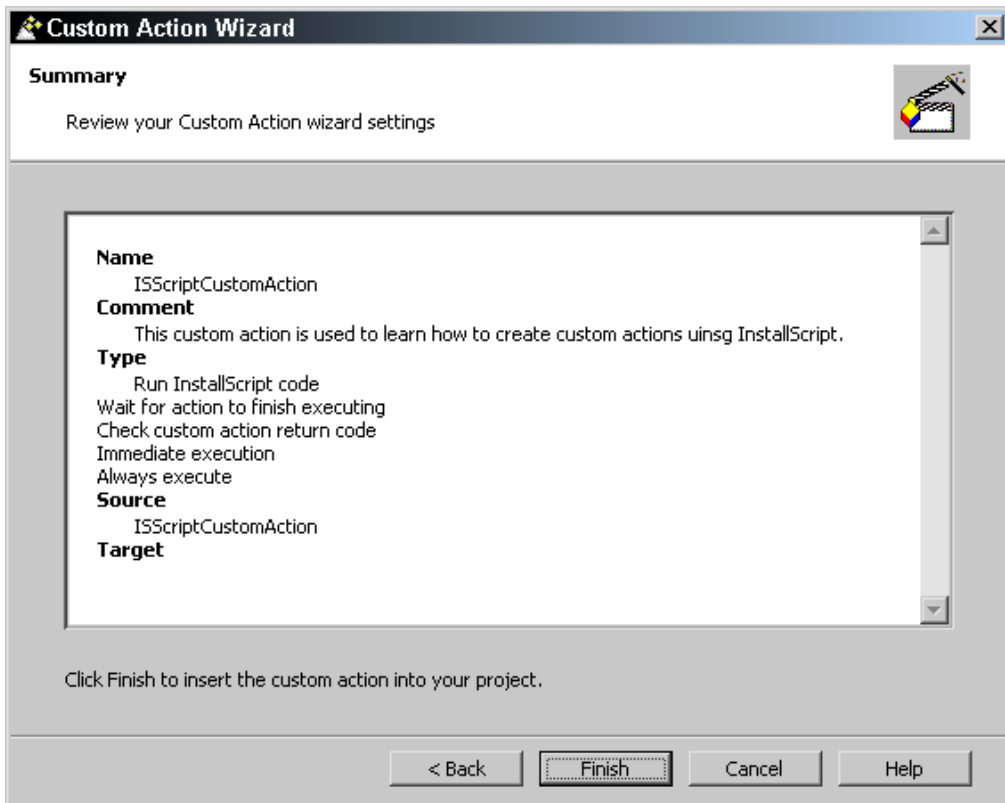


Figure 11-14: *The Summary panel of the Custom Action Wizard.*

Click Next to display the Summary panel (Figure 11-14). This panel shows all the options that you have selected in the Custom Action Wizard. You have the opportunity to go back and make any changes that are necessary. Then click Finish on the Summary panel to create the custom action that will call the `ISScriptCustomAction` function. Now, you need to rebuild the tables by selecting Build Tables Only from the Build drop-down menu on the Toolbar. This ensures that the entry made in the Setup.ini file is not destroyed.

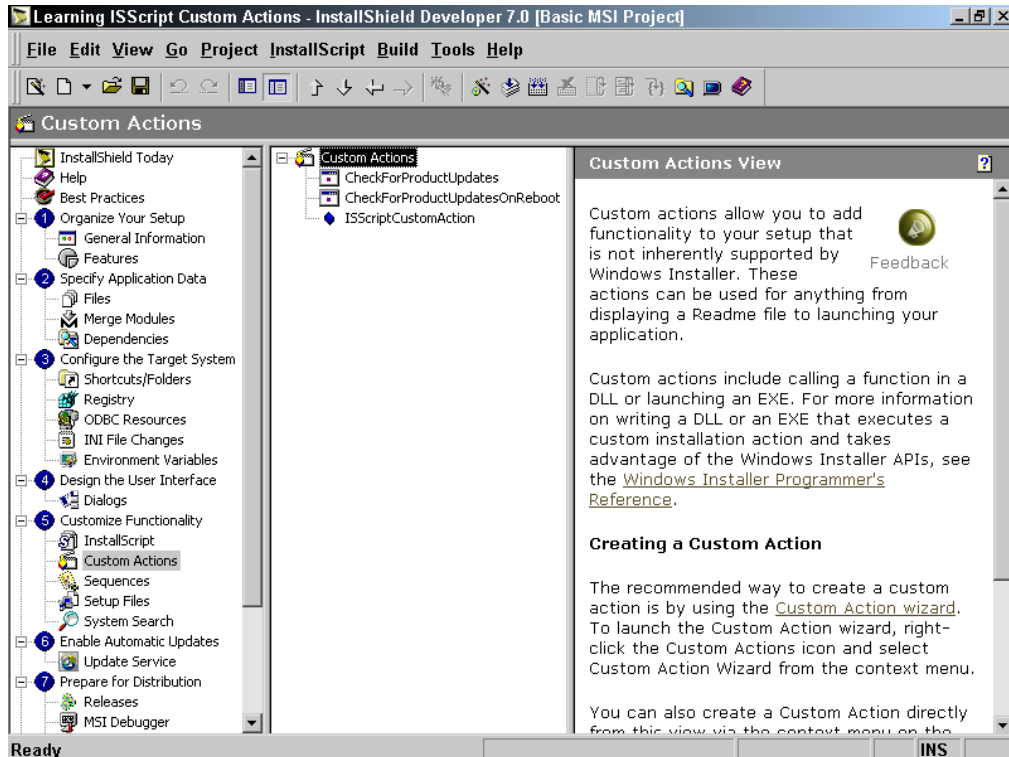


Figure 11-15: *The Custom Actions list.*

After the build is complete, note that there is more than one custom action in the list of custom actions (Figure 11-15). In Figure 11-15 you see the custom action that you just created and you see two other custom actions listed as well. These other two custom actions are always there and have to do with the InstallShield Update service. By default InstallShield Update services are enabled unless you specifically go to the Options dialog and disable them.

There are a number of additional custom actions that are added to the project the first time you create an InstallScript custom action. These custom actions do not show up in the Custom Actions view. You can only see them in the Direct Editor by going to the CustomAction table. You can, however, see them in the Sequences view where they have been inserted into the user interface and the execute sequence tables.

Using a Custom Action

In a Basic MSI project, you can insert an InstallScript custom in either the user interface sequence table or the execute sequence table. You can also attach an InstallScript custom action to a button in an authored dialog through the use of the DoAction control event.

In a Standard project you can use an InstallScript custom action custom only in the execute sequence table. The reason for this is discussed in Chapter 4. Because this example uses a Basic MSI project, this chapter covers the possible places that you can use an InstallScript custom action.

The InstallScript custom action that you have just created is an immediate custom action so you can test this by placing it in the user interface sequence. Place the custom action directly after the LaunchConditions action in the InstallUISequence table. To do this:

1. Expand the User Interface node under the Installation folder in the Sequences view.
2. Right-click on the LaunchConditions action and select Insert This displays the Insert Action dialog (Figure 11-16).
3. Select the ISScriptCustomAction from the list of custom actions and click OK.

The default for the Insert Action dialog is to display the custom actions that are available and not already inserted into the selected sequence table. You could have added a condition to this custom action by entering it into the Condition edit field but it is not required for this example. The comment that appears in the Comment edit field is the one you entered when you created the custom action in the Custom Action Wizard.

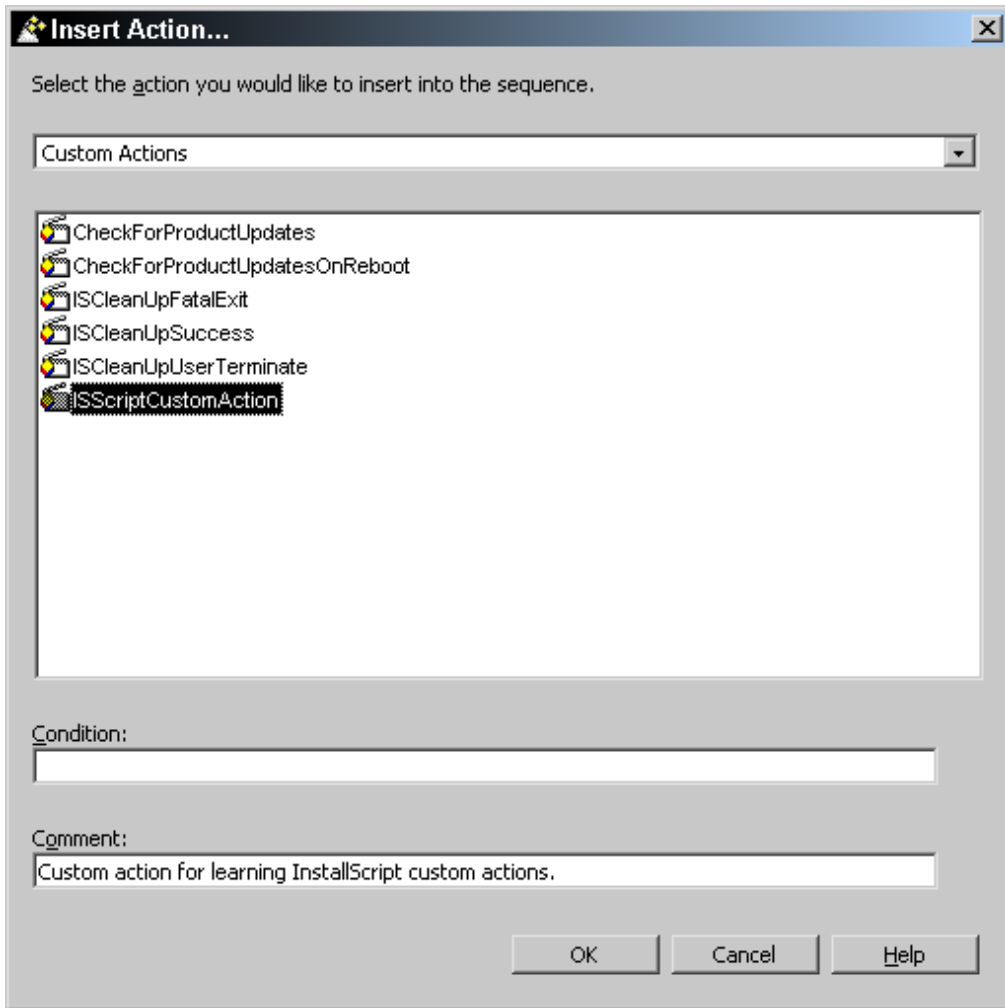


Figure 11-16: *The Insert Action dialog.*

When you insert a custom action in a sequence table as described above, it is inserted after the action or dialog that was selected when the Insert Action dialog was launched. The sequence number that the custom action is given is half way between the sequence numbers of the actions that come directly before and directly after. In this example, the sequence number will be 75 because the LaunchConditions standard action has a sequence number of 50 and the SetupInitialization dialog has a sequence number of 100.

Testing the Custom Action

To test this example, you need to build and run the program. To do this:

1. Build this initial example by selecting Build Tables Only from the Build drop-down menu on the Toolbar.
2. You can test that your custom action displays the message box by clicking the Run button on the Toolbar. When you run the installation program, a message box appears (Figure 11-17).

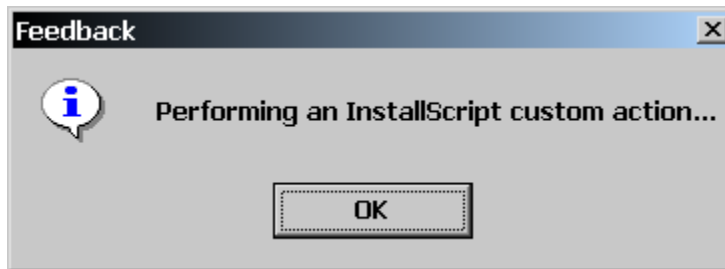


Figure 11-17: *The message box displayed by the `ISScriptCustomAction` custom action.*

3. Click OK on the message box. The installation continues to the end and displays the SetupCompleteSuccess dialog
4. Click Finish to remove this dialog from the screen.

The application is not registered on the system so you can continue to run tests without having to bother with an uninstallation between each test. The next section discusses how to create an InstallScript custom action without using the Custom Action Wizard.

An Alternate Way to Create a Custom Action

When you launched the Custom Action Wizard you saw that there were a number of other options on this context menu as shown in Figure 11-8. If you had decided to create the InstallScript custom action by selecting the first option on this context menu, you would have been presented with a property sheet where you would have to set all the parameters for the custom action. You can see what this property sheet looks like by going back to the Custom Actions view and clicking on the ISScriptCustomAction. You should see what is shown in Figure 11-17.

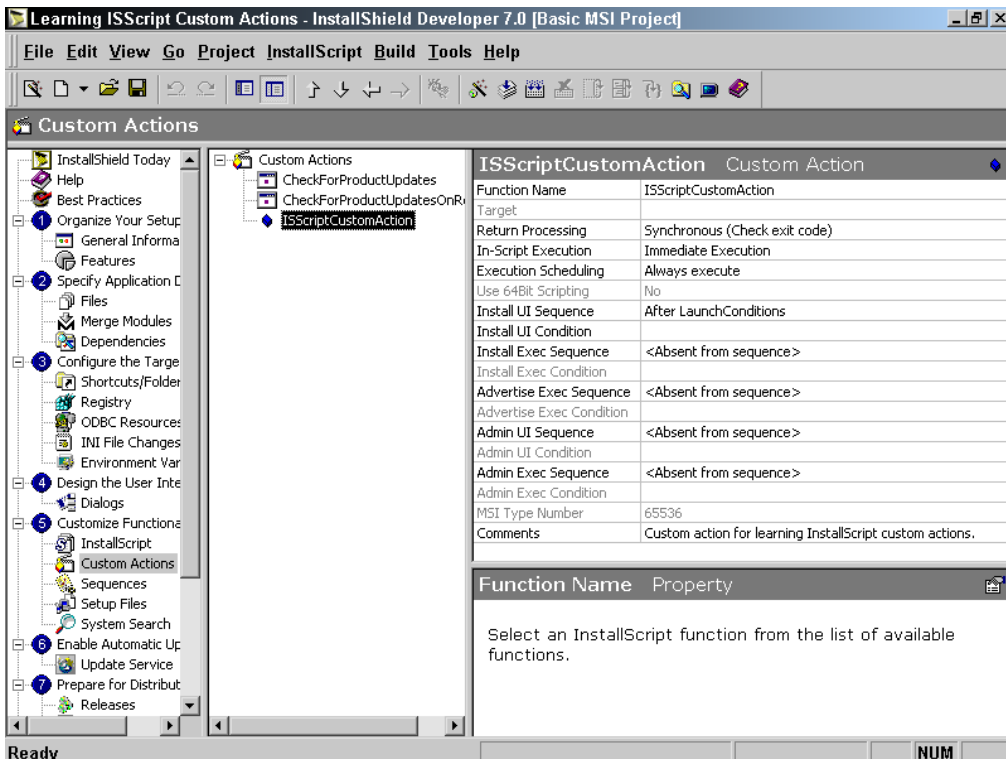


Figure 11-17: The property sheet for the ISScriptCustomAction.

As you can see in Figure 11-17 the property sheet for a custom action provides a location where you can do everything. You can create the custom action, set its options, and even place it into a sequence table. After using the Custom Action Wizard a few times you will probably want to use this approach since it is faster and you can see everything about the custom action in one glance.

Getting and Setting Properties

One of the most common operations that you need to carry out during an installation is to set the value of a property in the Property table and to retrieve the value of a property in the Property table. Chapter 3 explained that the properties in the Property table serve as the global variables used by the Windows Installer. Because properties are so important, the Property table is the only table where the Windows Installer offers two functions that set and get properties. As we will see later in this chapter, access to any other table requires the use of SQL.

The Windows Installer functions that are used to get and set properties in the Property table are `MsiGetProperty` and `MsiSetProperty`. The prototype for the `MsiGetProperty` function is shown below:

```
UINT MsiGetProperty{
    MSIHANDLE hInstall,    // handle to installer session
    LPCTSTR   szName,     // name of property
    LPTSTR    szValueBuf, // buffer for returned property value
    DWORD     *pchValueBuf // character count buffer
};
```

The prototype for the `MsiSetProperty` function is shown below:

```
UINT MsiSetProperty{
    MSIHANDLE hInstall,    // handle to installer session
    LPCTSTR   szName,     // name of property
    LPTSTR    szValue     // property value
};
```

Both of these functions take a handle to the installer session, as is the case with most of the database functions that are used in custom actions. Both of these functions require the name of the property whose value is either being retrieved or set. The name of a property is case sensitive.

When you use the `MsiGetProperty` function to retrieve a property's value, you need to know the size of the string buffer that will be used to hold the value of the property. This can be a problem because the value of a property can be unlimited in size. In order for this to work properly, you should call the `MsiGetProperty` twice, first with a zero length string for the buffer. When you do this, the last argument returns the length of the buffer required to hold the value of the property, not including the null terminator. You can then increment the size of the buffer by one and call the `MsiGetProperty` function a second time. The second time you call this function, it returns the value of the property.

In addition to setting the value of a property, you can also remove a property from the Property table using the `MsiSetProperty` function. To remove a property, pass the name of an existing property to the `MsiSetProperty` function and give it a NULL value.

One of the unique and valuable aspects of using InstallScript to implement custom actions is that you have access to the database in both immediate mode and deferred mode. This is not the case with any other type of custom action. For a non-InstallScript custom action, extra effort is required to make property values available to a deferred custom action. Deferred custom actions not created using InstallScript can only access properties that have been written into the Windows Installer execution script. To do this requires the creation of another custom action, which forces the property value or values to be written into the installation script. Retrieving the value of these property values can be extra effort depending on whether the value of more than one property needs to be obtained in deferred mode. Since this book only covers InstallScript custom actions this subject is not covered. The reasons why an InstallScript custom action can access the running database even in deferred mode is covered in Chapter 4.

Another benefit of using InstallScript to implement custom actions is that two InstallScript custom actions can communicate with each other through the use of global variables. Other types of custom actions need to use the Property table to communicate with each other. This means that the non-InstallScript custom actions need to use the `MsiGetProperty` and `MsiSetProperty` functions to pass values between themselves.

Example of Retrieving a Property Value

This simple example retrieves and displays the value of the DATABASE property (Figure 11-19).

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates the creation
//                of a simple custom action that retrieves and
//                displays the value of a property.
//
////////////////////////////////////
#include "isrt.h"
#include "iswi.h"

#define CAPTION    "Feedback"

export prototype ISScriptCustomAction(HWND);
////////////////////////////////////
// Function:      ISScriptCustomAction
//
// Purpose:       This function is the target of an InstallScript
//                custom action that gets and displays the value
//                of the DATABASE property.
////////////////////////////////////
function ISScriptCustomAction(hMSI)
STRING  svPropertyValue;
LONG    nvBufferSize;
begin

    // Set buffer to a NULL string and buffer size to zero.
    svPropertyValue = "";
    nvBufferSize = 0;

    // Make first call to get the actual buffer size
    // required for the DATABASE property.
    MsiGetProperty(hMSI, "DATABASE", svPropertyValue, nvBufferSize);

    // Increment size to account for NULL terminator
    // and then resize the string buffer.
    nvBufferSize++;
    Resize(svPropertyValue, nvBufferSize);

```

Figure 11-19: *Setup.rul* for a custom action that retrieves a value from the Property table.

```

// Make second call to get the actual value
// of the DATABASE property.
MsiGetProperty(hMSI, "DATABASE", svPropertyValue, nvBufferSize);

// Display the value of the DATABASE property.
sprintfBox(MB_OK, CAPTION, "DATABASE property = %s",
           svPropertyValue);

end;

```

Figure 11-19: *Continued.*

This example makes two calls to the `MsiGetProperty` Windows Installer API. The first call determines the size of the string buffer required to hold the value of the `DATABASE` public property and the second call retrieves the value of this property. Between the two calls, the program increments the required size of the string buffer to account for the `NULL` terminator and then sets the size of the string buffer. The final action is to display the value that is retrieved from the Property table.

Note that you do not need to make these two calls to the `MsiGetProperty` function if you are certain that a property value is not more than 1024 characters in length. As discussed in Chapter 8, this is the default size provided to any string passed to a DLL function. This is for backward compatibility with older scripts. However, it is best to get into the habit of making the two calls as shown in the above example.

You do not have to make any changes in your project since it already contains a custom action that targets the `ISScriptCustomAction` `InstallScript` function. Since the `ISScriptCustomAction` custom action is already inserted in the `InstallUISequence` table, you do not have to do anything with regard to its location. All you have to do is change the code, compile it, and then run the installation to see it work. You can either type in this code or you can copy it from the CD-ROM at the back of the book.

Note that the example in Figure 11-19 uses the `hMSI` session handle that is passed to the `InstallScript` function. You can also use an `InstallScript` system constant that contains the handle to the Windows Installer session. The name of this `InstallScript` system constant is `ISMSI_HANDLE`. This system constant is useful if you want to access the session handle in a function where it is not passed as one of the arguments. An example of this is in the `OnBegin` and the `OnEnd` event handlers, which are the only two event handlers supported in a Basic MSI project. These two event handlers are discussed later in this chapter.

Example of Setting a Property Value

When you first set up the Learning ISScript Custom Actions project, you placed a condition on a number of standard actions and on the InstallWelcome dialog in the InstallUISequence table. This eliminated the user interface and prevented the application from being registered so uninstallation is not required to test a custom action. The condition that you applied was a property named `CONDITION` that does not have a value.

In this example, you will check to see if the `CONDITION` property has a value and, if it does, this property is removed from the Property table (Figure 11-20). If the `CONDITION` property does not have a value, you will give it a value.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:       This script demonstrates the creation
//                 of a simple custom action that
//                 sets the value of a property.
//
////////////////////////////////////

#include "isrt.h"
#include "iswi.h"

#define CAPTION    "Feedback"

export prototype ISScriptCustomAction(HWND);

////////////////////////////////////
// Function:      ISScriptCustomAction
//
// Purpose:       This function is the target of an InstallScript
//                 custom action that sets the value of the
//                 CONDITION property if it does not already exist
//                 and removes it if it exists.
////////////////////////////////////
function ISScriptCustomAction(hMSI)
STRING svPropertyValue;

```

Figure 11-20: *Setup.rul showing the setting and removal of a value for a property.*

```

LONG    nvBufferSize;
begin

    // Set buffer to a NULL string and buffer size to zero.
    svPropertyValue = "";
    nvBufferSize = 0;

    // Make first call to get the actual buffer size
    // required for the DATABASE property.
    MsiGetProperty(ISMSI_HANDLE, "DATABASE", svPropertyValue,
                  nvBufferSize);

    // Check to see if the property already exists
    // by seeing if the buffer size is greater than zero.
    if(nvBufferSize > 0) then
        // Remove the CONDITION property if it exists.
        svPropertyValue = "";
        MsiSetProperty(ISMSI_HANDLE, "CONDITION", svPropertyValue);
    else
        // Set the value of the CONDITION property
        // if it does not exist.
        svPropertyValue = "1";
        MsiSetProperty(ISMSI_HANDLE, "CONDITION", svPropertyValue);
    endif;
end;

```

Figure 11-20: *Continued.*

The first thing that you do in this example is to check if the property exists by determining if the size of the required string buffer is greater than zero. If the required size is zero, you know that the property does not exist. Based on the assessment of the existence of the property, the example code then gives the property a value of "1" or sets its value to NULL, which removes it from the Property table.

Once again you do not have to do anything with the custom action itself. All you need to do is change the code and compile it. If you want to test this custom action for the scenario when the CONDITION property already exists, you need to enter this property and give it a value by using the Property Manager. After adding the CONDITION property to the Property table, build the project using the Build Tables Only option on the Build drop-down menu. The use of the Property Manager is discussed in Chapter 5.

Accessing Database Tables

The last section discussed how to get and set properties in the Property table. In order to do this for any other table in the MSI database, you need to use SQL and create views of the table. Then you work with the records in the view that you have created. You will be working with the SELECT SQL statement in this chapter.

During the running of an installation, you can access the database. Because the database is read-only, you can make only temporary changes and these changes are never persisted. When you work with the tables in the MSI database, there are a number of standard operations that you need to carry out in any custom action that you create. These steps are discussed in the following list.

1. The first operation that you need to do is to obtain a handle to the active database. The active database is the one being used for the current installation. To obtain the handle to the active database, call the `MsiGetActiveDatabase` function. This function takes only one argument and that is the handle to the Windows Installer session.
2. The next operation that is necessary is the creation of a view object. This is where you use the SQL query statement that will create the desired view. A view object is created using the `MsiDatabaseOpenView` function. This function takes the handle to the active database and the SQL query statement and returns in another argument the handle to the view object.
3. In order to fetch records from the view object, you need to execute the view. Execute the view object by calling the `MsiViewExecute` function. This function takes a handle to the view object as an argument and an optional argument that this chapter will not be using.
4. After performing the above three operations, you can fetch records from the view and read the individual fields that make up these records. A record in a view is obtained by calling the `MsiViewFetch` function. You pass a handle to the view object to this function and get back in another argument a handle to the fetched record.

5. A field in a record can be one of three data types: an integer, a string, or a binary stream. You can directly read a field if it is an integer or a string. If it is a binary stream, however, you have to handle this data type differently. For now, you will work only with fields that are integers or strings. To read a field that contains an integer value, use the `MsiRecordGetInteger` function and to read a record field that contains a string, use the `MsiRecordGetString` function. Each of these functions takes a handle to the record and the number of the field for which the value is being retrieved. For an integer value, the return value of the `MsiRecordGetInteger` function is the value in the record field. When retrieving the value of a string, you have to pass a buffer to receive the value of the string field and you also have to pass a size of the buffer. When getting the value of a string field in a record, you will want to use the same approach as used when getting the value of a property and that is we will want to call the `MsiRecordGetString` function twice. Call it the first time to get the true size of the buffer needed and then call it the second time with the correctly sized buffer.

When you want to only retrieve records from a table, you do not need to perform the alternate steps 4 and 5 below. The alternate steps 4 and 5 shown below are required only when adding temporary rows to a database table:

4. Once you have created a view object, you need to create a record object with the correct number of fields in it for the table to which you want to add it. To create a record object, use the `MsiCreateRecord` function. The only argument that this function takes is the number of the data fields to be contained in the record. This function returns a handle to a record object with the requested number of fields. To set a record field to an integer or string value, use either the `MsiRecordSetInteger` or `MsiRecordSetString` functions. Both of these functions take as arguments the handle to the record object, the field number to receive the value, and the value to be inserted into the field. You can also insert a file into a record field as a binary stream using the `MsiRecordSetStream` function. In this function, instead of providing a value as a third argument, you provide the path to the file that is to be streamed into the record field.

- Once you have created the record that you want to add to a particular table, you need to use the `MsiViewModify` function to add this record to the view. This function takes as arguments the handle to the view, an indicator of the modify mode, and the handle to the record being added to the view. When you are working with a running database during an installation, there is only one modify-mode that can be used and that is defined by the `MSIMODIFY_INSERT_TEMPORARY` constant.

The best way to understand the required operations is to take a look at an example of reading all the values from a table.

Example of Reading Values in a Table

Now that you are reading the values from an entire table, you need a better method than a message box to display these values. Accordingly, this example creates an approach to printing to a text file the rows in a database table as long as the table has all columns containing string values. The `InstallScript` function for printing to a text file uses the `FileSystemObject` object that is discussed in detail in Chapter 9. The code for this complete example is provided in Figure 11-21.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates how
//                 to work with tables and to print out
//                 the value in the fields of each row
//                 in the table.
//
////////////////////////////////////

#include "isrt.h"
#include "iswi.h"

#define CAPTION      "Feedback"
#define ForAppending 8 // Required for OpenTextFile method.
#define TemporaryFolder 2 // Required for GetSpecialFolder method.

```

Figure 11-21: *Setup.rul* for writing the rows of a table to a text file.

```

// Prototype of a function that is the target of a custom action.
export prototype ISScriptCustomAction(HWND);

// Prototype of a private function.
prototype INT PrintRecord(BYVAL INT, BYVAL INT, BYVAL STRING,
                          BYVAL INT);
////////////////////////////////////////////////////////////////
// Function:    ISScriptCustomAction
//
// Purpose:     This function is the target of an InstallScript
//              custom action that prints the columns in a
//              specified table in the database.
////////////////////////////////////////////////////////////////
function ISScriptCustomAction(hMSI)
STRING  szTableName, szQuery, szFileName, szColNames(3);
INT     i, nFields;
LONG    hDatabase, hView, hRecord;
OBJECT  fso, fldr, txtfile;
begin

    // Set the table name and the number
    // and name of the columns.
    szTableName = "ActionText";
    nFields = 3;
    szColNames(0) = "Action";
    szColNames(1) = "Description";
    szColNames(2) = "Template";

    // Define the query to be used to create the view.
    szQuery = "SELECT * FROM " + szTableName;

    // Get the handle to the active database.
    hDatabase = MsiGetActiveDatabase(hMSI);

    // Check for success of getting the database handle.
    if(hDatabase = 0) then
        SprintfBox(MB_OK, "Error", "Unable to get handle to " +
                  "active database.");
        return ERROR_INSTALL_FAILURE;
    endif;

    // Open a view using the SQL query statement.
    if(MsiDatabaseOpenView(hDatabase, szQuery, hView)
       != ERROR_SUCCESS) then
        SprintfBox(MB_OK, "Error", "Unable to " +
                  "open the view for the %s table.", szTableName);
        return ERROR_INSTALL_FAILURE;
    endif;

```

Figure 11-21: *Continued.*

```

// Execute the view just opened.
if (MsiViewExecute(hView, 0) != ERROR_SUCCESS) then
    sprintfBox(MB_OK, "Error", "Unable to execute " +
        "the view for the %s table.", szTableName);
    return ERROR_INSTALL_FAILURE;
endif;

// Create the text file name and location.
// The location is the Temp folder.
szFileName = szTableName + ".txt";
try
    set fso = CreateObject("Scripting.FileSystemObject");
    set fldr = fso.GetSpecialFolder(TemporaryFolder);
    szFileName = fldr ^ szFileName;
    set txtfile = fso.CreateTextFile(szFileName, TRUE);
catch
    return ERROR_INSTALL_FAILURE;
endcatch;

txtfile.WriteLine("This file contains the rows in the " +
    szTableName + " table.");

// Write a carriage return and line feed.
txtfile.WriteLine;

// Write the column names into the text file.
for i=0 to nFields-1
    txtfile.Write(szColNames(i));

    if(i < nFields) then
        txtfile.Write("\t");
    endif;
endfor;

// Write a carriage return and line feed
// and close the text file.
txtfile.WriteLine;
txtfile.Close;

// Fetch each record and print it out to the text file.
while (MsiViewFetch(hView, hRecord) = ERROR_SUCCESS)
    PrintRecord(hRecord, hView, szFileName, nFields);
endwhile;

//Close all handles
MsiCloseHandle(hRecord);
MsiCloseHandle(hView);
MsiCloseHandle(hDatabase);

```

Figure 11-21: *Continued.*

```

    return ERROR_SUCCESS;
end;

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Function:    PrintRecord
//
// Purpose:    This function prints to a text file the values
//             of all fields in a specified record.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
function INT PrintRecord(hRecord, hView, szFileName, nFields)
INT    i, nBufSize, nValue;
STRING szValue;
OBJECT fso, tso;
begin
    try
        // Create a FileSystemObject object and open the
        // text file that is passed to the function.
        set fso = CreateObject("Scripting.FileSystemObject");
        set tso = fso.OpenTextFile(szFileName, ForAppending, FALSE);
    catch
        return ERROR_INSTALL_FAILURE;
    endcatch;

    // Loop through the fields of the record and
    // write them into the text file.
    for i=1 to nFields
        nBufSize = 0;
        szValue = "";
        MsiRecordGetString(hRecord, i, szValue, nBufSize);
        nBufSize++;
        Resize(szValue, nBufSize);
        MsiRecordGetString(hRecord, i, szValue, nBufSize);
        tso.Write(szValue);

        // Add a tab delimiter between fields.
        if(i < nFields) then
            tso.Write("\t");
        endif;
    endfor;

    // Write a carriage return
    // and then close the file.
    tso.WriteLine;
    tso.Close;

    // Destroy the FileSystemObject object.
    set fso = NOTHING;
end;

```

Figure 11-21: *Continued.*

This example shows how to access the tables in a running database and also gives a generic approach to printing values from a custom action. This example accesses the ActionText table and prints out the values in each of the three columns that make up this table. A tab delimiter separates the values in each column.

In the function that is the target of the custom action, the program defines the table for which you want to create a view and creates an array that holds the column names for this table. The program then defines your SQL query that does a SELECT on all columns of the specified table. Following this, the program gets the handle to the active database, opens a view, and executes the view.

In the example's custom action function, you create the text file to which you will write all the rows of the ActionText table. Part of the text file creation is to write an initial line in the file telling which table is being written and you also write the column names to this text file. A tab delimiter also separates the column names. The program then closes this file and reopens it for each row that is written to it by the PrintRecord function.

Finally, the program loops through and fetches each record in the view, and passes the record handle to the PrintRecord function. This private function then opens the text file, writes the values of each column to the file, and closes the file. The looping continues until the MsiViewFetch Windows Installer function returns that there are no more records. Then the custom action returns and the installation process ends.

You do not have to do anything to get this custom action to run except compile the InstallScript code. The custom action that you have already created still targets the function that is used in this example. As an experiment, you can re-run the Custom Action Wizard on the ISScriptCustomAction custom action and change it to a deferred type. You can also use the property sheet for the custom action to change it to a deferred custom action. You can then place this redefined custom action in the InstallExecuteSequence table directly after the InstallInitialize action and select Build Tables Only from the Build drop-down menu on the Toolbar. When you run the installation in this configuration, the text file is created again just as it was when the custom action is run in immediate mode. This would not be possible with any other type of custom action. Only an InstallScript custom action has access to the database in deferred mode.

In this section you saw how to work with integer or string data in a table column. The next section takes a close look at how to work with binary data in a database column.

Streaming Out Binary Data

There are several tables in an MSI database that have a column that holds data of the binary data type. Binary data is a file that has been streamed into the appropriate field of a table. The table of most interest for us at this time is the Binary table. The Binary table holds a number of files that pertain to running InstallScript custom actions, as discussed in Chapter 4. Also in the Binary table are all the bitmaps and icons used in the user interface of a Basic MSI project.

An operation that is commonly handled in a custom action is the streaming out of the Binary table a file that was streamed in at build time. To perform this type of operation, you need to do a few new things in your script. The complete code for streaming out binary data is shown in Figure 11-22.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates how
//                to stream out binary data from the
//                Binary table.
////////////////////////////////////
#include "isrt.h"
#include "iswi.h"
#include "winapi.h"

#define CAPTION      "Feedback"
#define TemporaryFolder 2 // Required for GetSpecialFolder method.
#define CREATE_ALWAYS 2 // Required for the CreateFileA function.
// Prototype of a function that is the target of a custom action.
export prototype ISScriptCustomAction(HWND);

// Windows function prototypes
prototype KERNEL32.WriteFile(BYVAL LONG, BYVAL BINARY, BYVAL LONG,
                              BYREF LONG, BYVAL LONG);

```

Figure 11-22: *Setup.rul demonstrating how to stream out a file from the Binary table.*


```

////////////////////////////////////
// Function:      ISScriptCustomAction
//
// Purpose:       This function is the target of an InstallScript
//                custom action that streams out a file from
//                the Binary table.
////////////////////////////////////
function ISScriptCustomAction(hMSI)
STRING  szTableName, szBinaryKey, szQuery;
STRING  szFileName, bStream;
LONG    hDatabase, hView, hRecord, hFile;
INT     nBufSize, nWritten, nAttr;
OBJECT  fso, fldr;
begin

    // Set the table name and the number
    // and name of the columns.
    szTableName = "Binary";
    szBinaryKey = "NewBinary5";
    szFileName = "Welcome.bmp";

    // Define the query to be used to create the view.
    szQuery = "SELECT * FROM " + szTableName + " WHERE Name='" +
                szBinaryKey + "'";

    // Get the handle to the active database.
    hDatabase = MsiGetActiveDatabase(hMSI);

    // Check for success of getting the database handle.
    if(hDatabase = 0) then
        SprintfBox(MB_OK, "Error", "Unable to get handle to " +
                    "active database.");
        return ERROR_INSTALL_FAILURE;
    endif;

    // Open a view using the SQL query statement.
    if(MsiDatabaseOpenView(hDatabase, szQuery, hView)
        != ERROR_SUCCESS) then
        SprintfBox(MB_OK, "Error", "Unable to " +
                    "open the view for the %s table.", szTableName);
        return ERROR_INSTALL_FAILURE;
    endif;

    // Execute the view just opened.
    if(MsiViewExecute(hView, 0) != ERROR_SUCCESS) then
        SprintfBox(MB_OK, "Error", "Unable to execute " +
                    "the view for the %s table.", szTableName);

```

Figure 11-22: *Continued.*

```

        return ERROR_INSTALL_FAILURE;
    endif;

    // Fetch the record from the view just executed.
    if(MsiViewFetch(hView, hRecord) != ERROR_SUCCESS) then
        sprintfBox(MB_OK, "Error", "Unable to fetch the record " +
            "from the view for the %s table.", szTableName);
        return ERROR_INSTALL_FAILURE;
    endif;

    // Define file to receive the binary stream.
    set fso = CreateObject("Scripting.FileSystemObject");
    set fldr = fso.GetSpecialFolder(TemporaryFolder);
    szFileName = fldr ^ szFileName;

    hFile = CreateFileA(szFileName, GENERIC_WRITE, 0, 0,
        CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0);

    nBufSize = 1023;
    nAttr = 0;
    // Continue to read the binary data
    // until nothing is left.
    while (nBufSize > 0)
        // Read the stream into a buffer, 1023 bytes at a time
        // with one byte for the null terminator.
        MsiRecordReadStream(hRecord, 2, bStream, nBufSize);

        if(nBufSize > 0)then
            //Write the buffer to a file.
            WriteFile(hFile, bStream, nBufSize, nWritten, nAttr);
        endif;
    endwhile;

    // Destroy the FileSystemObject object
    // and the folder object.
    set fldr = NOTHING;
    set fso = NOTHING;

    //Close all handles
    CloseHandle(hFile);
    MsiCloseHandle(hRecord);
    MsiViewClose(hView);
    MsiCloseHandle(hView);
    MsiCloseHandle(hDatabase);

    return ERROR_SUCCESS;
end;

```

Figure 11-22: *Continued.*

To write binary data to a file, you need to use two Windows APIs that are not part of the normal functions available in InstallScript. These two functions are the `CreateFile` and the `WriteFile` functions. For the `CreateFile` function, you need to call the ANSI version, thus the call will be to the `CreateFileA` function. The prototype of this function is provided in the `winapi.h` header file that you need to add at the top of your script. For a full description of the `CreateFile` function, refer to the MSDN library.

The `WriteFile` function needs to be explicitly prototyped at the top of your script. This function is exported by `KERNEL32.DLL` so you need to use the `DLL` name as part of the prototyped. Note that the second argument has a data type specified as `BINARY`. The use of this special designation prevents the InstallScript engine from converting the `STRING` data type that is passed to the multi-byte character set. Instead, the string that is passed is kept in its raw form, which is just a stream of bytes. A full description of the `WriteFile` function appears in the MSDN library.

The particular binary data that you are going to stream out of the `Binary` table is the bitmap that is used in the `InstallWelcome` dialog and a few others. The key to this particular binary data is "NewBinary5". To retrieve just this row from the `Binary` table, your SQL query statement needs to be more complex. You need to include the `WHERE` clause and specify that the `Name` column of the `Binary` table be equal to "NewBinary5". The `name` column of the `Binary` table is the primary key for this table and it is a string data type. In your query string, you need to surround the variable holding the name of the "NewBinary5" key with single quotes. Thus we have a SQL query constructed as follows in your script.

```
szQuery = "SELECT * FROM " + szTableName + " WHERE Name='" +
          szBinaryKey + "'";
```

Notice that there are single quotes inside the double quotes on either side of the `szBinaryKey` variable name.

This example performs the same database access operations as the previous example, but here you do not have to fetch more than one record so there is no looping involved. Either the record is in the `Binary` table or it is not. If the record is not present, the program terminates the installation by returning the `ERROR_INSTALL_FAILURE` constant to the Windows Installer.

The program uses the `FileSystemObject` object to create the absolute path to the file that will be used to contain the binary data streamed out of the `Binary` table. The name used for the file to be created is `Welcome.bmp`. To create this file, you need to use the `CreateFileA` function since the `FileSystemObject` can create only text files.

Finally, the program uses the `MsiRecordReadStream` Windows Installer function to stream out the binary data and the `WriteFile` Windows API function to write this data into the `Welcome.bmp` file. The `MsiRecordReadStream` function operates in such a way that it returns in one of its arguments the number of bytes that it has streamed out of the record. The approach that you need to take in streaming out data is to choose some size that you will stream out and then loop through the `MsiRecordReadStream` function until it returns a zero as the number of bytes that have been read. When this occurs, you know that all the binary data has been extracted. This works because the `MsiRecordReadStream` function picks up after each call where it left off with the previous call.

The size that you use for streaming out the binary data is the default size of a string when it is passed to a DLL function. This size is 1024 bytes and you pass 1023 bytes as the size of the buffer to the `MsiRecordReadStream` function since you need to reserve one byte for the null terminator. During the looping process, you call the `WriteFile` function as long as the returned size is greater than zero. Once the returned value is zero bytes, the program is finished so it closes the file and closes all the handles that were created to access the database. The program returns `ERROR_SUCCESS` to the Windows Installer to indicate that the custom action was successful and that the installation can continue.

As shown in the code in Figure 11-22, the `Welcome.bmp` file is created in the temporary folder defined by the operating system. You can go there after the custom action and the installation are complete to find this file. When you open this file in Microsoft Paint, the bitmap that is used as the background for the `InstallWelcome` dialog is displayed.

The OnBegin and OnEnd Event Handlers

As discussed in Chapter 4, the `OnBegin` event handler is called as part of the initialization process and the `OnEnd` event handler is called as part of the uninitialization process. The `OnBegin` event handler is called only once. It is called in the `InstallUISequence` table when there is a Full or Reduced user interface level being used for the installation and it gets called in the `InstallExecuteSequence` table when a Basic or None user interface level is being used. The same is the case for the `OnEnd` event handler. It is also only called once at the end of an installation.

The default implementation of the `OnBegin` and `OnEnd` event handlers is a no-op and, as such, they perform no actions. You would use these event handlers to perform your own initialization and uninitialization operations necessary for your installation program. During the linking process, the code you write replaces the default implementation. There are no arguments passed to the `OnBegin` and `OnEnd` event handlers so if you need to do something with the active database, you need to obtain the handle to the active session by using the `ISMSI_HANDLE` system constant.

As an experiment, you can add a message box to the `OnBegin` and the `OnEnd` event handlers and see where in the installation process these message boxes are displayed. As part of this experiment, remove the `ISScriptCustomAction` custom action from where it is inserted in the sequence tables and delete it totally from the project.

To delete a custom action:

1. Go to the Custom Actions view under Step 5.
2. Expand the Custom Actions tree.
3. Right-click on the custom action to be deleted, and select Delete.

For the best effect, you should also go to the Property Manager and set a value for the `CONDITION` property. This allows you to run a full installation with the full user interface. The code for this experiment is shown in Figure 11-23.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:      This script demonstrates the
//                OnBegin and the OnEnd event handlers.
//
////////////////////////////////////

#include "isrt.h"
#include "iswi.h"

#define CAPTION      "Feedback"

function OnBegin
begin

    sprintfBox(MB_OK, CAPTION, "Executing the OnBegin " +
                "event handler");

end;

function OnEnd
begin

    sprintfBox(MB_OK, CAPTION, "Executing the OnEnd " +
                "event handler");

end;

```

Figure 11-23: *Setup.rul for demonstrating the OnBegin and the OnEnd event handlers.*

To build this example, select `Build Tables Only` from the `Build` drop-down menu on the `Toolbar`. When you run this example, the message box in the `OnBegin` event handler is displayed before any other internal dialog is displayed in the `InstallUISequence` table. Also note that the message box in the `OnEnd` event handler is displayed in the `InstallExecuteSequence` table before control is returned to the `InstallUISequence` table.

This experiment demonstrates that if you need to perform additional operations in the `InstallUISequence` after changes have been made to the target system, you cannot use `InstallScript` custom actions. This is because, as part of the process of executing the `OnEnd` event handler, the `ISMSiServerStartup` custom action initiates the shut down of the `InstallScript` engine running in the `IDriver.exe` process.

Using the `MsiDoAction` Function

The `MsiDoAction` Windows Installer function is what the Windows Installer uses to execute the actions and/or dialogs that it encounters in the sequence tables. You can also use this function inside a custom action to run actions and/or dialogs without these actions or dialogs being inserted into a sequence table. An action that is run in this fashion has to be one of the defined standard actions or a custom action that is defined in the `CustomAction` table. To display a dialog using this function, the dialog must be defined in the `Dialog` table. However, a dialog cannot be displayed in the execute sequence table, only in the user interface sequence table.

To understand this better, you can implement a simple example where, with your `ISScriptCustomAction` custom action running in immediate mode, you create a deferred custom action that gets written into the installation script generated by the Windows Installer. You do not have to place this custom action into the `InstallExecuteSequence` table, but do have to define it in the `CustomAction` table. The code for this example is shown in Figure 11-24.

```

////////////////////////////////////
//
// File Name:      Setup.rul
//
// Description:    Learning the InstallScript language
//
// Comments:       This script demonstrates how
//                 to use the MsiDoAction Windows Installer
//                 function.
//
////////////////////////////////////

#include "isrt.h"
#include "iswi.h"

```

Figure 11-24: *Setup.rul* for demonstrating the use of the `MsiDoAction` function.

```

#define CAPTION      "Feedback"

// Prototype of functions that are the targets of a custom action.
export prototype ISScriptCustomAction(HWND);
export prototype CreateDeferredCustomAction(HWND);

/////////////////////////////////////////////////////////////////
// Function:      ISScriptCustomAction
//
// Purpose:      This function is the target of an InstallScript
//                custom action that creates a deferred custom action.
/////////////////////////////////////////////////////////////////
function ISScriptCustomAction(hMSI)
begin

    SprintfBox(MB_OK, CAPTION, "Executing MsiDoAction");

    MsiDoAction(hMSI, "CreateDeferredCustomAction");

    return ERROR_SUCCESS;
end;

/////////////////////////////////////////////////////////////////
// Function:      CreateDeferredCustomAction
//
// Purpose:      This function is the target of an InstallScript
//                deferred custom action that displays a message box.
/////////////////////////////////////////////////////////////////
function CreateDeferredCustomAction(hMSI)
begin

    SprintfBox(MB_OK, CAPTION, "Running a deferred custom action " +
                "that was created by a call to the MsiDoAction function.");

    return ERROR_SUCCESS;
end;

```

Figure 11-24: *Continued.*

This example is just for demonstration purposes and all that these two custom actions do is to display message boxes. The `ISScriptCustomAction` custom action is an immediate type and is the custom action that you have been running throughout this chapter. The new custom action that you create in this example and that targets the `CreateDeferredCustomAction` function is the deferred type. You create this custom action using the Custom Action Wizard, but you do not insert this custom action into the `InstallExecuteSequence` table. You can use the name of the

function as the name of the custom action, just as you did with the `ISScriptCustomAction` custom action.

Before you build your project, you need to insert the `ISScriptCustomAction` custom action in the `InstallExecuteSequence` table after the `InstallInitialize` action. This is where the Windows Installer starts to create the installation script. If the `ISScriptCustomAction` custom action is placed before the `InstallInitialize` action, a Windows Installer run-time error occurs. Build your project by selecting `Build Tables Only` from the `Build` drop-down menu.

When you run the installation, a message box from the immediate custom action informs you that the program is about to run the `MsiDoAction` function. The program then runs the `MsiDoAction` function and this writes your deferred custom action into the installation script. When the Windows Installer executes the installation script, a message box informs you that a deferred custom action is running.

Conclusion

This chapter introduced the creation of custom actions using `InstallScript`. The discussions used a Basic MSI project but a Standard project can use `InstallScript` custom actions in the execute sequence table. You learned that `InstallScript` has some distinct advantages over other approaches to creating custom actions. One of the major advantages of `InstallScript` is that you can share data between two or more custom actions through the use of global variables. This is not possible with other types of custom actions. Other types of custom actions need to use the `Property` table in order to share data. Another advantage of `InstallScript` is that it has access to the database even in deferred mode. This is something that no other type of custom action can do.

Chapter

12

User Interface Basics

Chapter 10 covered a number of common tasks that you perform to create installation projects. One of the interesting things about all these tasks is that there is no difference between how you implement them in a Standard project or a Basic MSI project. Now we come to the subject that defines the real difference between Standard and Basic MSI projects. This difference is how the user interface is displayed during an installation is implemented.

In a Standard project, the user interface is created programmatically in a somewhat similar fashion as is done in a Windows application. With a Basic MSI project, you describe the user interface by adding rows to a number of tables in the database and the Windows Installer is responsible for turning this information into the user interface. Regardless of how you define the user interface for an installation, the Windows operating system carries out the same operations behind the scenes to

display the various dialogs. To be able to work with the user interface, particularly in a Standard project, you need to understand what the Windows operating system is doing when it displays a dialog on the screen. Because of this, this chapter begins with a short review of the Windows mechanism for handling a dialog window.

The Basics of Windows Dialogs

The first thing to understand is that a dialog box is a window and is handled in a similar fashion as any other window in an application. A dialog is used to interact with the end user in order to solicit input and provide feedback. This interaction is done through the use of controls, which are windows that are children of the dialog window.

There are two types of dialogs, modal and modeless. A modal dialog is one that takes the focus and does not release this focus until the dialog is dismissed. This means that, until the end user responds to the dialog in some way and dismisses it, nothing else can happen in the process that launched the dialog. A modeless dialog has the opposite functionality. When a modeless dialog is displayed, it is possible for other actions to take place in the same process. This is why the normal user interface sequence for an installation is comprised of modal dialogs, but the dialog that displays the progress of the installation is a modeless dialog.

Defining a Dialog

A dialog is different than a normal window in that you have to describe the dialog in a way that tells Windows what the dialog should look like. A dialog box and the controls that compose the functionality of the dialog box are described in templates. A template is a specification that defines the height, width, style, and the controls that make up a dialog box. A dialog box template is binary data. Windows either loads this binary data into memory from a resource or it is created directly in memory by the application. A dialog resource can be contained in a separate file or it can be part of another executable file. Regardless of how the template is created, one has to be supplied by the application for Windows to be able to display the dialog.

The standard approach to defining a dialog is to create a resource that is either contained in an application's executable file or is in a special resource dynamic link

library. A resource starts out as a script that is contained in a file that has a .rc extension. Though it is possible to create a resource script using a text editor, it is better to use a resource editor or a dialog editor to perform this activity. A dialog editor will create this resource script for you based on how you construct the dialog. The resource script is then compiled into an object file that has a .res extension. Finally the resource object file is incorporated into an executable or a dynamic link library during the linking process.

A dialog is defined inside a resource script using the DIALOG or the DIALOGEX statements. Following one of these statements are other statements that define the features found in the dialog box. These particular statements are option statements. Option statements specify the style, caption, and font that are to be used when the dialog box is displayed. Finally, after the option statements comes a block of script that is called the resource definition body. It is in the resource definition body that the controls to be included on the dialog as child windows are defined. The general form of a dialog definition in a resource script file is as follows:

```
<Dialog name or dialog ID> DIALOGEX x, y, width, height
STYLE <Style parameters OR'd together>
CAPTION "<Caption of dialog>"
FONT <Font statement parameters>
<Other option statements as required>
BEGIN

    <Control resource statements and associated parameters>

END
```

In the above general form for defining a dialog, note that you can use either a string to name a dialog or you can identify the dialog with an ID that is a 16-bit integer number. After the unique identification of the dialog comes the DIALOGEX (or DIALOG) statement followed by the location of the upper-left corner of the dialog when it is displayed, and then the height and width of the dialog.

Following the DIALOGEX statement come the option statements. The ones that are shown are those that are typically used for dialogs created for an installation's user interface. No option statements are required. If no option statements are defined, default values are used. Below the option statements comes the script block where the controls to be included on the dialog are defined. This block of script is bounded

by the BEGIN...END keywords. It is also acceptable to use curly braces in place of these keywords to delimit this block of script.

The Dialog Controls

There are a number of controls that are typically used on a dialog box. The controls that are useful in creating a installation's user interface are described in the following list.

Push button: A push button is a rectangle containing text. The text indicates what action is to be taken when the user clicks the button. A push button can be either a standard button or a default button. A standard push button is typically used to start an operation. It receives the keyboard focus when the user clicks it. A default push button, on the other hand, is typically used to indicate the most common or default choice. It is a button that the user can select by simply pressing the ENTER key when a dialog box is first displayed. There is also a push button that is called an owner-drawn button. This type of button is created by the application and has no predefined appearance or usage. Its purpose is to provide a button whose appearance and behavior is defined by the application alone.

Check box: A check box consists of a square box and a text label that indicates a choice the user can make by selecting the button. Check boxes are usually displayed in a group to permit the user to choose from a set of related, but independent options. A set of check boxes provides a *non-exclusive* set of choices.

A check box can be one of four styles: standard, automatic, three-state, and automatic three-state. Each style can assume two check states: selected with a check mark shown inside the box or cleared where there is no check mark inside the box. In addition, a three-state check box can assume an indeterminate state where there is a grayed box inside the check box. Repeatedly clicking a standard or automatic check box toggles it from selected to cleared and back again. Repeatedly clicking a three-state check box toggles it from selected to cleared to indeterminate and back again. With a standard style check box, the application has to handle setting and unsetting the check mark. When the style is automatic, the check box itself handles the setting or unsetting of the check mark.

Group box: A group box is a rectangle that surrounds a set of controls such as check boxes or radio buttons. A group box contains a text label in the upper-left

corner that describes the general purpose of the controls that it surrounds. The sole purpose of a group box is to organize controls related by a common purpose.

Radio button: A radio button consists of a round button and a text label. The text indicates a choice the user can make by selecting the button. An application typically uses radio buttons in a group box to permit the user to choose from a set of related options. A group of radio buttons provides the user with an *exclusive* set of choices.

Combo box: There are three types of combo boxes: a simple combo box, a drop-down combo box, and a drop-down list box. All combo boxes consist of two fields. There is a list field and a selection field. The list field displays the options that are available for the user to select and the selection field displays the current selection. With the simple combo box, the list is always displayed. With the other two types of combo boxes, the user has to click on an icon to the right of the combo box to display the list. For the simple and the drop-down combo boxes, the selection field is an edit field that allows the user to type in their own values. The selection field for the drop-down list is a static text field where it is not possible to type in a value that is not already in the list.

List box: There are two types of list boxes, single selection and multiple selection. A single selection list box allows the user to select only one item from the list box. The multiple selection list box allows the user to select more than one item from the control. You can define list boxes to have multiple columns and to have the items in the list box sorted.

List view: A list view control displays a collection of items, each of which consists of an icon and a label. It is also possible to have additional columns to the right of the icon and label that are used to display information about items in the control. A good example of the use of a list view control is the Windows control panel. A list view control can display its contents in four different ways, which are called views. These four ways are described in the following list.

- **Icon view:** In this type of view each item is a 32 x 32 icon with a label below it. The items can be dragged to other locations in the control.

- **Small icon view:** In this type of view each item is shown with a 16 x 16 icon with the label displayed to the right of the icon. The items can be dragged to other locations in the control.
- **List view:** In this type of view each item appears as a small icon with the label to the right. The items cannot be dragged to any other location in the view.
- **Report view:** In this type of view each item appears as a small icon with the label to the right in the first column. Additional information is arranged in columns to the right of the first column.

Edit control: This type of control is used to display text to the user and to allow the user to add or modify text. There are two types of edit controls, those that can handle only a single line of text and those that can handle multiple lines of text.

Static controls: Static controls come in three different forms. These are simple graphics controls, static image controls, and static text controls. A simple static graphics control displays a frame or a filled rectangle. If you decrease the height of a filled rectangle to 0, it becomes a line. A rectangle can be filled with white, gray, or black. A static image control is used to display bitmaps, icons, or enhanced metafiles.

The most common of all static controls is the text static control. This type of control is used for labels and instructions to the user. A text static control can display text as aligned left, aligned center, or aligned right. If a text static control is not large enough to hold all the text to be displayed, Windows cuts off the text at the end of the string that does not fit.

Progress bar: This type of control is used to indicate the progress of a lengthy operation such as one that occurs during an installation. This control consists of a rectangle that is gradually filled from left to right using the system highlight color. It is also possible to display text inside the progress control. The bar that is used to display the progress can be continuous or it can be displayed as blocks of color.

Tree control: A tree control is used to display a hierarchical list of items. Each item in a tree control consists of a label and an optional bitmapped image. Each

item can have a list of sub-items associated with it. When the user clicks an item, the associated list of sub-items can be expanded or collapsed.

Every control is identified by a 16-bit integer so that a dialog can tell which control the user is activating. Behind each dialog there is a function that responds to the actions that the user takes in the dialog. How a dialog function works is the topic of the next section.

Creating Dialog Functionality

As mentioned earlier all of a dialog box's functionality is contained in an associated function. Windows creates the dialog based on the dialog template that is defined. The dialog function and Windows send messages back and forth to implement the functionality of the dialog box. Figure 12-1 shows the communication between an application, its dialog function, and Windows. This figure shows the communication for a modal dialog box. This chapter focuses on modal dialog boxes because they are the most common type of dialog used in an installation.

Understanding the function that is called to create the dialogs in an installation is important only for obtaining a deeper understanding of what is going on behind the scenes. The Windows `DialogBox` macro or the `DialogBoxParam` function is used to create modal dialogs from a dialog template in a resource file. The Windows `DialogBoxIndirect` macro and the `DialogBoxIndirectParam` function are used to create modal dialogs from templates that have been created in memory.

The Windows `CreateDialog` macro and the `CreateDialogParam` function can also be used to create modeless dialogs from a dialog template in a resource file. The `CreateDialogIndirect` macro is used to create modeless dialogs from templates that have been created in memory. The same is true for the `CreateDialogIndirectParam` function. You can create a modal dialog using one of the macros or functions that are normally used to create a modeless dialog. To do this, wrap a call to the `CreateDialogIndirectParam` function in another function. The first thing that this wrapper function does is to disable the parent window. After the call to the `CreateDialogIndirectParam` function, a message loop is set up. This is similar to what a Standard project does to implement the script-based dialogs that are used for the user interface. In a Basic MSI project, the user interface is described completely in the database. Because of this, Windows

Installer uses one of the indirect functions to create the dialogs by first creating the template in memory then passing a pointer to the dialog template's location in memory.

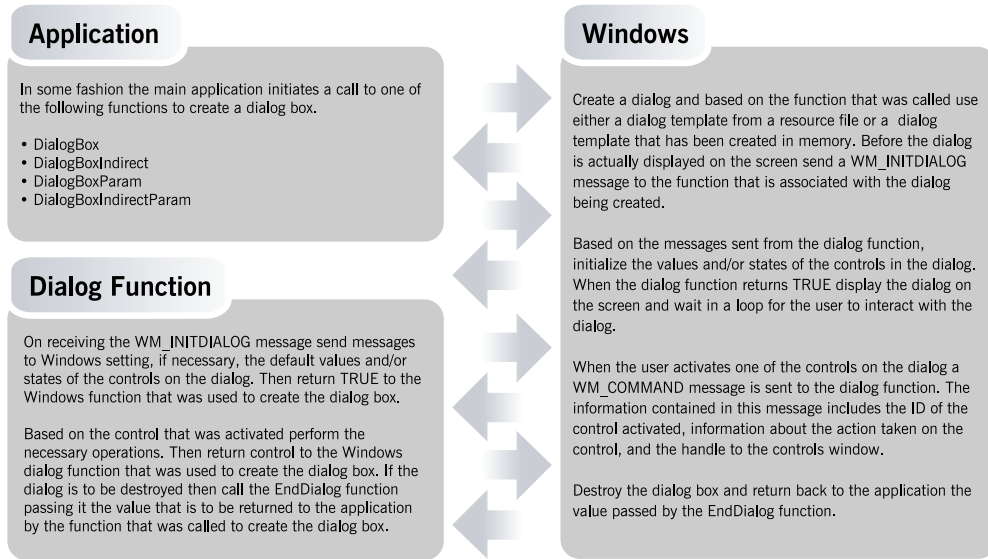


Figure 12-1: *Communication between Windows and a dialog function for a modal dialog.*

The message passing mechanism shown in Figure 12-1 is the same that is used in any Windows application. Windows displays the user interface of an application, but the code behind the user interface is implemented in special callback functions in the application. These callback functions are not called from within the application but directly by Windows. Note that there are no arrows showing any communication between the application itself and the dialog function that is implemented as part of the application code.

It is instructive to look at some pseudo code that demonstrates how a Windows application might implement a dialog function (Figure 12-2). You will do something similar when you implement a custom dialog in a Standard project.

```

// Call the DialogBox macro from the application.
DialogBox(hModule, "MyDialog", hwnd, DlgFunc);

// Definition of the function behind the MyDialog dialog.
BOOL CALLBACK DlgFunc(HWND hDlg, UINT iMsg,
                    WPARAM wParam, LPARAM lParam)
{
    switch(iMsg)
    {
        case WM_INITDIALOG:

            Perform initialization operations.
            return TRUE;

        case WM_COMMAND:
            // Get the ID of the control that was activated.
            switch(LOWORD(wParam))
            {
                case ID_BUTTON1:

                    Perform operations consistent with the user
                    clicking the button with the control ID of
                    ID_BUTTON1.

                    return TRUE;

                case ID_BUTTON2:

                    Perform operations consistent with the user
                    clicking the button with the control ID of
                    ID_BUTTON2.

                    return TRUE;
                case ID_LISTBOX:

                    Perform operations consistent with the user
                    clicking the button with the control ID of
                    ID_LISTBOX.

                    return TRUE;
                .
                .
                .
            } // End inner switch
        } // End outer switch

        return TRUE;
    }
}

```

Figure 12-2: *A sample dialog function showing the handling of messages.*

The first line of code in Figure 12-2 calls the `DialogBox` macro that is used to create the dialog. The first argument passed to this macro is a handle to the file that contains the resource describing the dialog. The second argument is the name of the dialog that is to be displayed. The third argument is a handle to the window that is the parent of the dialog that is going to be created. The fourth and final argument is the name of the dialog function that implements the dialog's functionality.

The remainder of the code in Figure 12-2 provides a look at what a dialog function will contain. When Windows calls the dialog function, it passes as the first argument a handle to the dialog that is being created. The second argument is an unsigned integer that indicates the message that is being sent to the function. The final two arguments hold the details of the message that must be interpreted in order to respond appropriately to the action the user took. Though there are other messages that might be sent to a dialog, the most common messages are defined by the `WM_INITDIALOG` and the `WM_COMMAND` message constants.

As is shown in Figure 12-1, the first message sent to a dialog function is the `WM_INITDIALOG` message where you can perform initialization operations if required. The dialog is not displayed on the screen until after the dialog function returns from performing the initialization operations.

After the dialog is displayed on the screen, Windows sends the `WM_COMMAND` message to the dialog function for any control that is activated by the user. Along with the `WM_COMMAND` message, the `wParam` and the `lParam` arguments hold information about what the user did in the dialog box. Both the `wParam` and the `lParam` arguments are unsigned integers. The `wParam` argument contains two pieces of information, one piece contained in the lower 16-bits and the other contained in the upper 16-bits of the argument. The `lParam` argument holds a handle to the control that was activated by the user.

Since the `wParam` argument contains two pieces of information, you have to extract the two pieces by using the `LOWORD` and the `HIWORD` macros. In the pseudo code shown in Figure 12-2 the statement `LOWORD(wParam)` is used as the argument to the inner switch statement. This extracts the ID of the control that was activated by the user. If you needed to see what the user did to the control, you could insert a statement inside the case statement for the activated control that retrieved the upper 16-bit portion of the `wParam` argument. This statement would look something like this:

```
notMsg = HIWORD(wParam);
```

The information contained in the upper 16 bits of the `wParam` argument is the *notification message*. Every control has one or more notification messages that Windows can send to the dialog function. Notification messages are defined by constants just the same as with the `WM_COMMAND` message constant. A message constant can be identified by the fact that it contains an "N" in the constant name. For example if the user double-clicks on a list box control item, Windows sends the `LBN_DBLCLK` notification message. You could then insert something similar to the following C language statement in your code where the control ID was equal to `ID_LISTBOX` as seen in Figure 12-2.

```
if(HIWORD(wParam) == LBN_DBLCLK)
    // Perform necessary operations.
```

This section has been a brief review of how Windows handles dialog boxes. This provides the background to understand how to create custom dialogs in a Standard project. The information in this section also provides an understanding of how the Windows Installer creates dialogs out of entries in a database. The next section discusses how a Standard project and a Basic MSI project differ in their approach to creating, modifying, and displaying dialog boxes.

Standard Projects vs. Basic MSI Projects

In a Standard project, you display dialog boxes by calling built-in `InstallScript` functions or `InstallScript` functions that you create for your custom dialogs. In a Basic MSI project, you can display dialogs by either placing the dialog name in a sequence table or making the dialog the target of a control event. Dialogs in a Basic MSI project are described in a number of tables in the MSI database.

This section examines the mechanics of working with dialogs in both Standard and Basic MSI projects.

The Dialogs View in Standard Projects

To modify and/or create dialog resources in a Standard project, use the Dialogs view under Step 4 in the View List. This view provides a complete list of all the dialogs that are part of a Standard project. The dialogs in this view that have names that start with "Sd" are dialogs that have the dialog function created using InstallScript. These dialogs are *script dialogs*.

The dialog names that do not start with "Sd" are termed *built-in dialogs*. Built-in dialogs also have their dialog functions implemented in InstallScript. The reason that you see these dialogs are called built-in because in the early days of InstallShield these particular dialogs actually were part of the scripting engine and could not be modified by the setup developer. Now these dialogs can be modified just as the Sd dialogs can be modified.

Regardless of where the dialog function is implemented, the dialog templates for both Sd dialogs and built-in dialogs are contained in a resource-only dynamic link library named `_isres.dll`. One of these files is available for each language that is installed using an InstallShield language pack. The English copy of this file is found in the following folder:

```
C:\Program Files\InstallShield\Developer\Redist\0409\i386\_isres.dll
```

If you have access to Microsoft Visual C++ you can actually open up the `_isres.dll` resource file for any language and view the dialogs. The folders under the `Redist` folder are each named using the hexadecimal representation of the language ID for the language of the `_isres.dll` resource file that is installed there. At build time the name of the resource file is modified to include the decimal representation the resource language's language ID. For example when an English-only project is built, there is one copy of the resource file and its name is `_ISRES1033.DLL`.

The Dialogs view for a Standard project is shown in Figure 12-3. This view provides a number of options for working with dialogs in a Standard project. The most important of these options is the Dialog Editor. The Dialog Editor is a visual resource editor similar to what you get in Visual Basic.

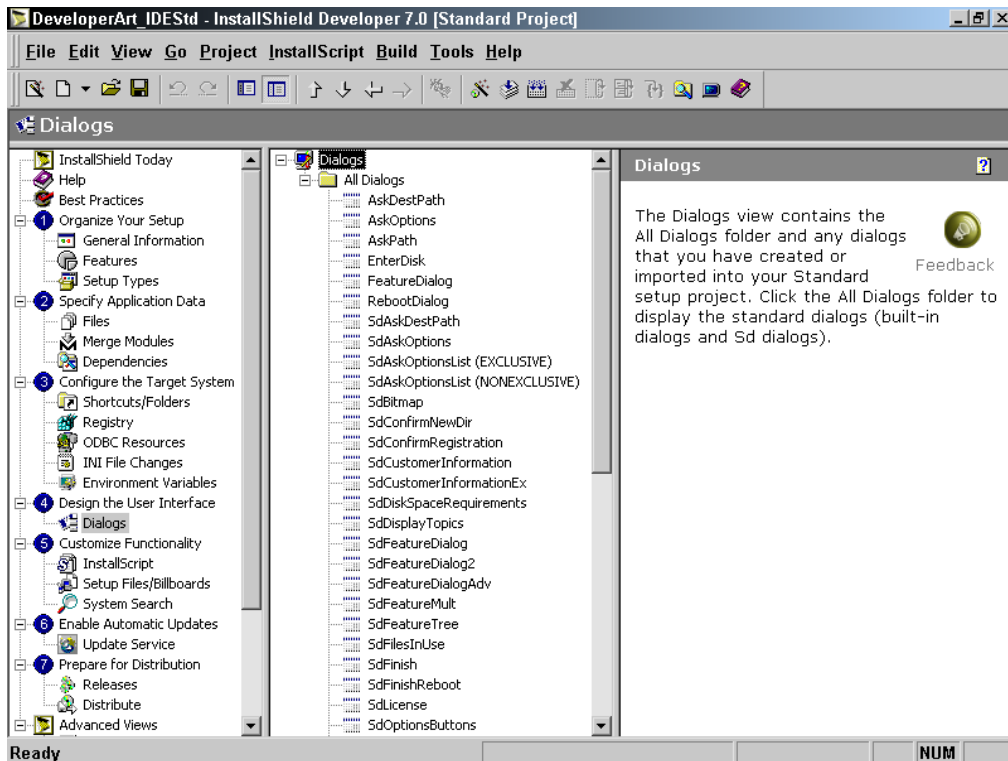


Figure 12-3: *Standard project Dialogs view.*

Click on the name of a dialog to display a representation of it in the right panel. Along with the picture of the dialog is a short description of the dialog's purpose and a link to the help topic for the dialog function.

The dialog functions that are used in InstallScript are different than what was described in the previous section. The previous section discussed how a dialog function would be created in a Windows application. The dialog functions created in InstallScript normally have arguments that allow you to customize some of the text that is displayed on the dialog, as well as to retrieve information from the user. However, the actual internal operation of a script dialog function is very much like what was discussed in the last section.

At the top of the list of dialogs shown in Figure 12-3 there is an All Dialogs folder icon. Right-click on this icon to display the context menu that is shown in Figure 12-4.

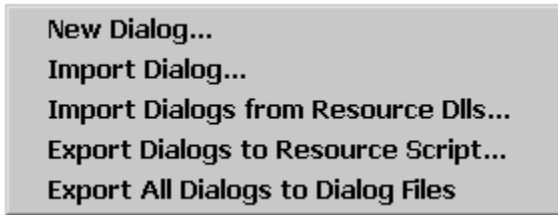


Figure 12-4: *The All Dialogs context menu.*

Each of the context menu options is discussed in the following list.

New Dialog: This option displays the New Dialog dialog (Figure 12-5). This dialog displays the Dialog Gallery, which offers a selection of exported dialogs along with a Blank Dialog and a template for a basic dialog.

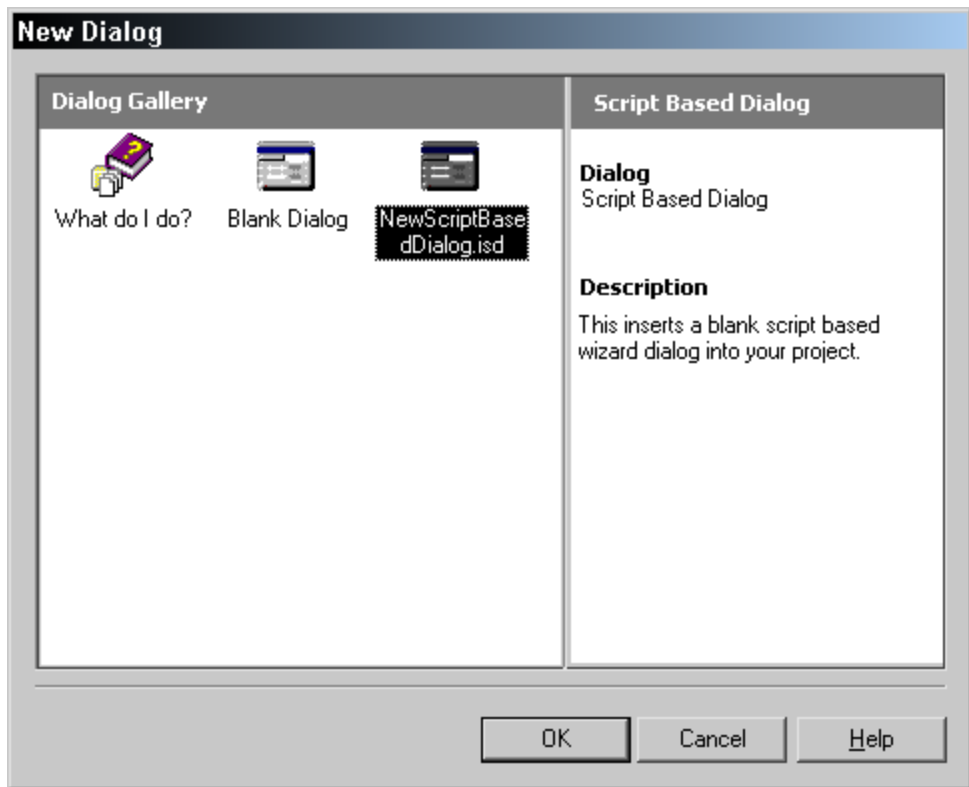


Figure 12-5: *The New Dialog dialog showing the Dialog Gallery.*

You can use an exported dialog or a blank dialog as the starting point for creating a custom dialog. However, you will normally want to use the `NewScriptBasedDialog.isd` file as your starting point. Because you have not exported any dialogs, only the Blank Dialog and the `NewScriptBasedDialog` appear in the Dialog Gallery. When you create a new dialog using the `NewScriptBasedDialog`, the appropriate rows are added to the Dialog table and other associated tables in the project file. You can view these entries by using the Direct Editor. This option from the Dialog Gallery provides a dialog that has the bitmap, branding, and the three buttons that are found on most dialogs. If you select Blank Dialog, one entry is made in the Dialog table and one entry is made in the Control table. The entry in the control table is a hidden text static control that is used to display dialog branding.

Import Dialog: This option allows you to browse to a dialog file and import it into your project. A dialog file is created when you export a dialog to a file with an `.isd` extension. A dialog file is a special InstallShield Developer file format that describes the components of a dialog that can be imported into a project for use in the Dialog Editor.

Import Dialogs from Resource Dlls: This option allows you to browse to a resource-only dynamic link library and import the contents of this DLL into your project.

Export Dialogs to Resource Script: A resource script is a text file with an `.rc` extension. You can use a resource script to create a resource-only dynamic link library. This option creates a resource script only for those dialogs that have been imported or dialogs that you are creating. This option cannot be used to create a resource script for any of the built-in or script dialogs that are part of a Standard project unless you first use the Edit option on the dialogs context menu.

Export All Dialogs to Dialog Files: This exports dialogs to an InstallShield Developer dialog file that has an `.isd` extension. This option is useful only for exporting dialogs that you have imported or are creating from a blank dialog. This option cannot be used to export to dialog files the built-in and script dialogs that are part of a Standard project unless you first use the Edit option on the dialogs context menu.

Right-click on a dialog to display the context menu for working with individual dialogs (Figure 12-6).

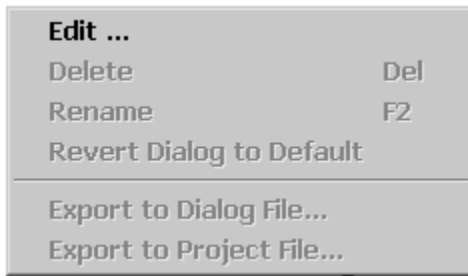


Figure 12-6: *The context menu for individual dialogs in a Standard project.*

Each of the options on the individual dialog context menu is discussed in the following list.

Edit Select this option to convert one of the built-in or script dialogs into a template that you can edit in the Dialog Editor. This option extracts the definition of the dialog from the `_isres.dll` resource file and places this definition into the tables in the project file that are concerned with the user interface. When the project is built, a new resource-only dynamic link library is created with the name `_ISUser<language id>.DLL`. To provide the functionality for the dialog, you need to add a new dialog function in `InstallScript`.

Delete: This option deletes any new dialogs that have been created. This option cannot be used to remove one of the dialogs that is included by default with a Standard project.

Rename: This option renames any new dialogs that have been created. This option cannot be used to rename the dialogs that are included by default with a Standard project.

Revert Dialog to Default: Use this option to revert back to the original default dialog definition that comes with a Standard project. The action of this option is to remove from the Direct Editor view all entries that have been made for this dialog when you selected the Edit command and any changes that were made subsequent to the Edit action.

Export to Dialog File: When you have edited one of the default dialogs or created a new dialog, you can export that dialog to an `.isd` file. This makes the dialog available to any other project that wants to use it.

Export to Project File: With this option you can export a dialog directly to another project. This option can only be used on dialogs that you have converted for editing purposes or new dialogs that you have created.

Now that we have discussed the facilities available in the Dialogs view, we need to look at how the compilation of a resource file is accomplished.

Compiling Resource Files for a Standard Project

When you create a new dialog or edit one of the default dialogs in a Standard project, you are creating entries in the user interface related tables that are turned into a resource-only dynamic link library at build time. The user interface related tables are the Dialog, Control, ISLocalDialog, ISLocalControl, RadioButton, ListView, Property, ListBox, TextStyle, Icon, Binary, and Checkbox tables. The name of the DLL that is created is `_ISUser<language id>.DLL` and this file is streamed into the Binary table of the database. When the installation is run, this file is streamed out to the SUPPORTDIR location and the absolute path for this file is provided by the ISUSER system variable. Just as is done with the `_isres.dll` resource DLL, the name of the file is streamed into the Binary table with the name modified using the decimal representation of the language ID. For the English projects discussed in this book, the name of the file as streamed into the Binary table is `_ISUser1033.dll`. When the file is streamed out to the temporary location during the installation, the name of the DLL does not include the language ID.

The process of creating a resource dynamic link library consists of three steps:

1. Create a resource script file from the information that is in the project file tables, visible in the Direct Editor view. A resource script file is a text file that has an `.rc` file extension. This file is created in the root folder of the build location of the associated project. For example, the location for this file for the Standard project that you used in Chapter 10 is as follows:

```
C:\MySetups\DeveloperArt_IDEStd
```

2. After the resource script file is created it has to be turned into a resource object file. This step is implemented through the uses of the resource

compiler RC.EXE. The resource compiler creates a file with a .res file extension. This object file is created in the same folder as shown in Step 1. The resource compiler is shipped with InstallShield Developer and is located in the following folder:

C:\Program Files\InstallShield\Developer\Script\Resource

3. The final step is to create the resource-only dynamic link library from the object file created in Step 2. This operation is carried out by the linker utility LINK.EXE. This utility is also shipped with InstallShield Developer and is located in the same folder as the resource compiler. The resource dynamic link library is created in the same folder on the build machine as both the resource script file and resource object file.

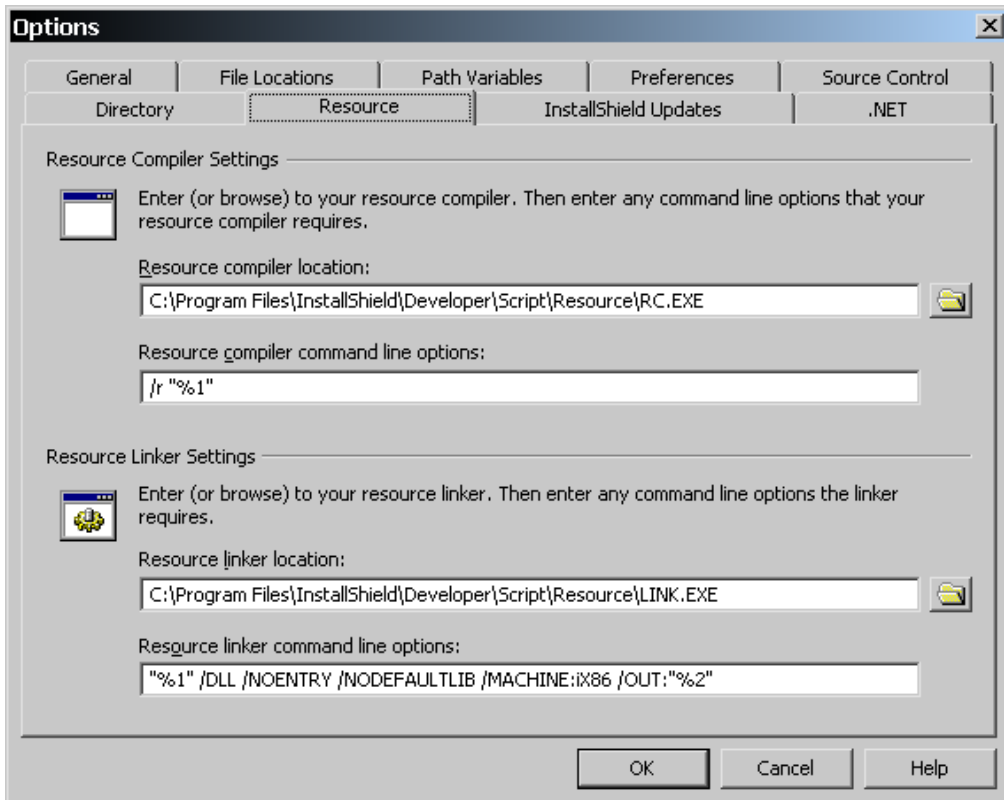


Figure 12-7: *The Resource property sheet of the Options dialog.*

Though InstallShield Developer ships the resource compiler and linker utility that also ships with Microsoft Visual C++, you can use a different resource compiler and/or linker utility. However, it is strongly recommended that you use the Microsoft resource compiler. Modification of the resource compiler can be performed in the Options dialog launched from the Tools pull down menu. This dialog is shown in Figure 12-7.

In the top half of the Resource tab (Figure 12-7), you can define the location on the build machine of the resource compiler. You can also specify the command line options to be used when the resource script is compiled into the resource object file. In the bottom half of the tab, you can specify the location of the linker utility and the command line options to be used.

The default command line options tell the linker to create a resource-only DLL that has no other libraries linked in, and that the target machine will be an Intel compatible platform. The final argument specifies the name of the DLL that is to be created.

The Dialogs View in Basic MSI Projects

The Dialogs view in a Basic MSI project (Figure 12-8) lists dialogs that differ considerably from what is shown in Figure 12-3. A Basic MSI project does not deal with resource scripts and resource DLLs. The only thing that you have to do is to make entries in a database that will be used by the Windows Installer to dynamically create dialog templates in memory so they can be displayed. Creating simple dialogs in a Basic MSI project is faster than in a Standard project. The drawback is that there is much less flexibility in the dialogs you can create in a Basic MSI project.

In the DeveloperArt_IDEMSI.ism project file right click on the All Dialogs folder icon at the top of the list of dialogs to display the same context menu as shown in Figure 12-4. The purpose and functionality of each of the items on the context menu is the same. Click a dialog name in the list to see a description of the major attributes of the dialog in the right panel. Links in the description panel allow you to move to the two nodes that appear under each of the dialogs in the list.

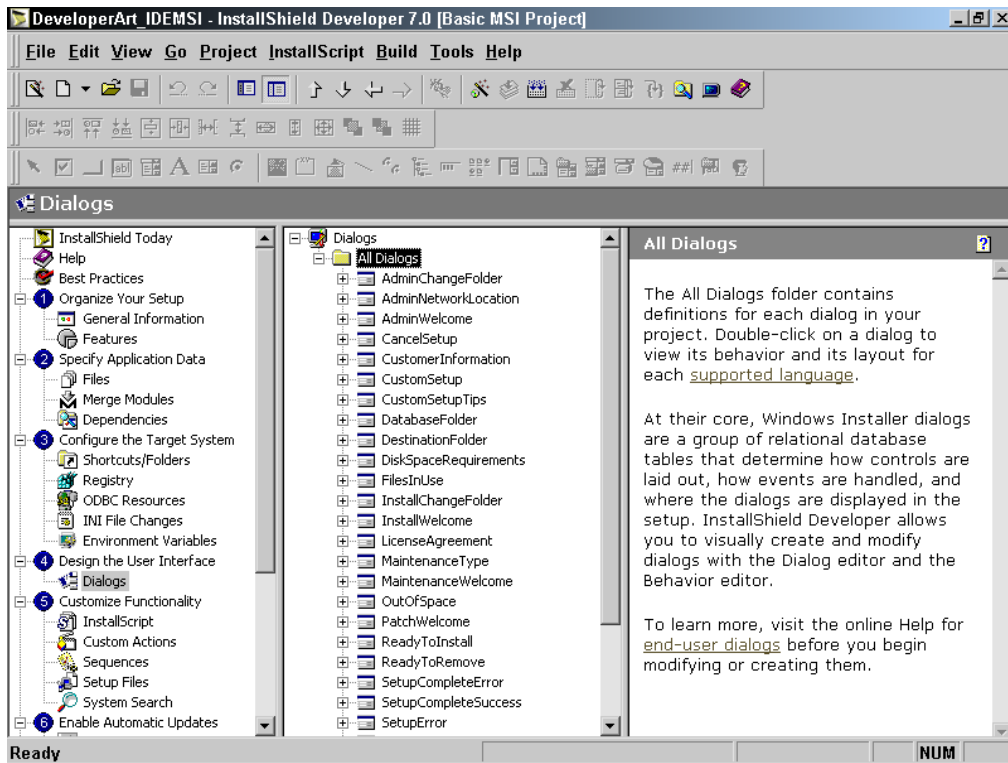


Figure 12-8: *The Dialogs view in a Basic MSI project.*

Right-click on a dialog in the list to display the individual dialog context menu for Basic MSI dialogs (Figure 12-9). The first two options on this context menu allow you to delete or rename a dialog in the list. The bottom two options are the same as the options on the Standard dialog context menu (Figure 12-6). You can export the selected dialog to a dialog file with an .isd extension and/or export the dialog into another project.

Under each dialog in the list there is a Behavior node and a language node. Click on the language node, in this case English (United States) to open the Dialog Editor where you can modify the layout of existing Basic MSI dialogs. Click on the Behavior node to open an editor where you can define the functionality of the dialog's controls.

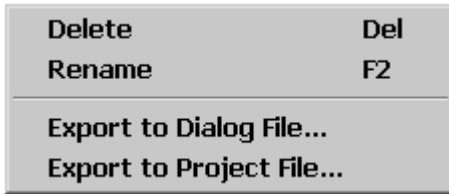


Figure 12-9: *The context menu for individual dialogs in a Basic MSI project.*

There are three elements to defining the functionality of a control on a dialog in a Basic MSI project. These consist of Control Events, Subscriptions, and Control Conditions.

Control Events: A control event is used to define the action taken when the user interacts with a control on a dialog. There is a predefined set of control events that are recognized by the Windows Installer and not all control events are usable with every one of the types of recognized controls. This is one of the limitations in a Basic MSI project with regard to developing a user interface. In a Standard project, you have the full flexibility that the InstallScript language offers.

Subscriptions: A subscription is when one control can respond to the action taken by another control. Normally this is a matter of the subscribing control changing the value of one of its attributes in response to the control event issued from another control. This particular functionality in Windows Installer is very limited and is used primarily to display progress messages and data in the progress dialog.

Control Conditions: A control condition is used to modify the behavior or appearance of a control based on the truth of a condition associated with the control. Based on a condition's value, a control can be hidden, shown, disabled, enabled, or made the default control on the dialog. The default control is the one that responds to the user pressing the Return key.

You will work with these behavior items later in this chapter. The next section provides some hands-on experience with working with the user interface in both types of projects. It begins with the user interface in a Standard project.

Generating a User Interface for a Standard Project

This section first discusses the use of the predefined built-in and script dialogs. It then takes a look at one of these predefined dialogs and examines the `InstallScript` dialog function. Finally, it walks through an example of creating a simple custom dialog along with its `InstallScript` dialog function.

The basis for discussing the user interface in a Standard project is the Developer Art project first created in Chapter 5 and modified in the examples of Chapter 10. The name of this project is `DeveloperArt_IDEStd.ism`.

Understanding the Default User Interface

To display a Standard project's user interface, you need to call the `InstallScript` dialog functions from the `OnFirstUIBefore` and `OnFirstUIAfter` control events when performing a fresh installation of the application. When performing a maintenance installation, call the `InstallScript` dialog functions from the `OnMaintUIBefore` and `OnMaintUIAfter` control events. Normally all the dialogs except for the dialog announcing the end of the installation process are called from the `OnFirstUIBefore` and `OnMaintUIBefore` event handlers.

When you create a Standard project without using the Project Wizard, a default version of the `OnFirstUIBefore` event handler is provided (Figure 12-10). Looking at the default version of this event handler provides a good example of the correct approach to developing a user interface using the predefined dialogs. This event handler has been reformatted for presentation purposes.

Immediately after the `begin` statement is a set of statements that are commented out by default. Because these lines of code are commented out, the installation runs only with a wizard user interface. There is no background window created for the installation on which the user interface dialogs are displayed.

The `SetTitle` function is used to manipulate the title displayed in the background window for an installation. If a background window is not enabled then this function

has no effect. The `SetTitle` function can be used to set the title in the title bar of the background and it can also be used to set the size and color of the title that is displayed in the upper left hand corner of the background window. The `Enable` function is a multi-purpose function that is used to enable many things in an installation. With regard to the user interface in a Standard project this function is used to define the background window as being a normal window or a maximized window. It is also used to define that there is to be a background window used for the installation. Finally, the `SetColor` function is used to define the color that is used to fill the background window. Optionally you can modify the default font style using the `SetFont` function. You can also use this function to define a different font typeface for the text of the background title.

```

////////////////////////////////////
//
// FUNCTION:   OnFirstUIBefore
//
// EVENT:     FirstUIBefore event is sent when installation is run
//            for the first time on given machine. In the handler
//            installation usually displays UI allowing end user
//            to specify installation parameters. After this
//            function returns, ComponentTransferData is called
//            to perform file transfer.
//
////////////////////////////////////
function OnFirstUIBefore()
    NUMBER nResult, nSetupType, nvSize, nUser;
    STRING szTitle, szMsg, szQuestion, svName, svCompany, szFile;
    STRING szLicenseFile;
    LIST list, listStartCopy;
    BOOL bCustom;
begin

    // TO DO: if you want to enable background, window title,
    //         and caption bar title
    // SetTitle( @PRODUCT_NAME, 24, WHITE );
    // SetTitle( @PRODUCT_NAME, 0, BACKGROUNDCAPTION );
    // Enable( FULLWINDOWMODE );
    // Enable( BACKGROUND );
    // SetColor(BACKGROUND,RGB (0, 128, 128));

    SHELL_OBJECT_FOLDER = @PRODUCT_NAME;

    nSetupType = TYPICAL;

```

Figure 12-10: *The default implementation of the `OnFirstUIBefore` event handler.*


```

Dlg_SdWelcome:

    szTitle = "";
    szMsg   = "";
    nResult = SdWelcome(szTitle, szMsg);
    if (nResult = BACK) goto Dlg_SdWelcome;

    szTitle   = "";
    svName    = "";
    svCompany = "";

Dlg_SdCustomerInformation:

    nResult = SdCustomerInformation(szTitle, svName,
                                    svCompany, nUser);
    if (nResult = BACK) goto Dlg_SdWelcome;

Dlg_SetupType:

    szTitle = "";
    szMsg   = "";
    nResult = SetupType(szTitle, szMsg, "", nSetupType, 0);

    if (nResult = BACK) then
        goto Dlg_SdCustomerInformation;
    else
        nSetupType = nResult;
        if (nSetupType != CUSTOM) then
            nvSize = 0;
            FeatureCompareSizeRequired(MEDIA, INSTALLDIR, nvSize);
            if (nvSize != 0) then
                MessageBox(szSdStr_NotEnoughSpace, WARNING);
                goto Dlg_SetupType;
            endif;
            bCustom = FALSE;
            goto Dlg_SdStartCopy;
        else
            bCustom = TRUE;
        endif;
    endif;

Dlg_SdAskDestPath:

    nResult = SdAskDestPath(szTitle, szMsg, INSTALLDIR, 0);
    if (nResult = BACK) goto Dlg_SetupType;

```

Figure 12-10: *Continued.*

```

Dlg_SdFeatureTree:
    szTitle      = "";
    szMsg        = "";
    if (nSetupType = CUSTOM) then
        nResult = SdFeatureTree(szTitle, szMsg, INSTALLDIR, "", 2);
        if (nResult = BACK) goto Dlg_SdAskDestPath;
    endif;
Dlg_SdStartCopy:
    szTitle = "";
    szMsg   = "";
    listStartCopy = ListCreate( STRINGLIST );
    // The following is an example of how to add a
    // string(svName) to a list(listStartCopy).
    // eg. ListAddString(listStartCopy,svName,AFTER);
    nResult = SdStartCopy( szTitle, szMsg, listStartCopy );

    ListDestroy(listStartCopy);

    if (nResult = BACK) then
        if (!bCustom) then
            goto Dlg_SetupType;
        else
            goto Dlg_SdFeatureTree;
        endif;
    endif;

    // setup default status
    Enable(STATUSEX);

    return 0;
end;

```

Figure 12-10: *Continued.*

If you remove the comment characters from in front of the following lines of code, a full screen window with a green color background is displayed.

```

SetTitle( @PRODUCT_NAME, 24, WHITE );
SetTitle( @PRODUCT_NAME, 0, BACKGROUNDCAPTION );
Enable( FULLWINDOWMODE );
Enable( BACKGROUND );
SetColor(BACKGROUND,RGB (0, 128, 128));

```

If you build the project and run the installation, a user interface, as shown in Figure 12-11, appears. The background window has a title bar and the name of the product is shown in the title bar. The name of the product is also shown on the background

window in a white font. Because you have not changed the font face, Arial is used by default.

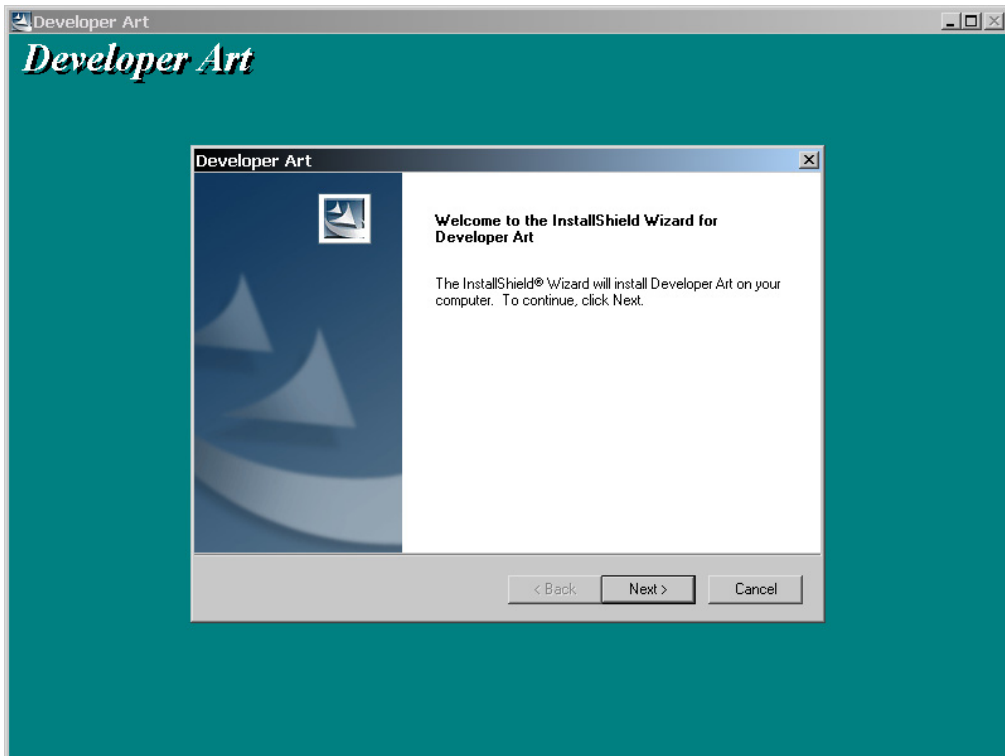


Figure 12-11: *The user interface with a full-screen background.*

If you want to have a background window without a title bar, leave the following line of code commented out as shown.

```
// Enable( FULLWINDOWMODE );
```

By default the title that is displayed comes from the value of the `PRODUCT_NAME` string table ID. This string table entry is created at build time and is not visible in the string table under the General Information view. If you want to display a different title, you could create a string table entry and then reference that in place of the `PRODUCT_NAME` entry. The `@` character is required in front of the string table ID in order to get the value of the string ID at run time. If you do not like the Arial typeface, you can change it using the `SetFont` built-in function.

The next line of code sets the value of the `SHELL_OBJECT_FOLDER` system variable to the value of the `PRODUCT_NAME` string ID. This system variable is not used by the default code in the `OnFirstUIBefore` event handler and it can be safely removed. The next line of code sets the default value for the setup type and is used in the `SetupType` dialog function to present a default choice to the end user.

The next section of the code implements the dialog sequence that is presented to the end user during the installation. The end user must be able to move forward and back through the set of dialogs that compose the information collection part of the user interface. To enable this functionality, your code will use the `goto` statement. Prior to the call to any `InstallScript` dialog functions, you need to place a label to which you can jump using a `goto` statement. Chapter 7 discusses the standalone `goto` statement and the special `if` statement that includes the `goto` as part of the statement. The flow of dialogs defined in the code (Figure 12-10) is represented in Figure 12-12.

Each of the dialog functions that are called in the default user interface has a number of arguments that are passed. In the default implementation many of these arguments are `NULL` strings, which causes the default values for those arguments to be used. For example, in the `SdCustomerInformation` dialog function, a `NULL` string passed for the `szTitle` parameter causes this function to display the default string "Customer Information" in the title bar. When `NULL` strings are passed for `svName` and `svCompany` parameters of the `SdCustomerInformation` dialog function, the values that are initially displayed to the end user are values that are found in the target machine's registry.

After the call to the `SetupType` dialog function (Figure 12-10), a comparison is made between the space required for installing the application and that available space on the target drive. This check is made explicitly if the setup type selected is not a custom setup. A check on space is also made when performing a custom setup, but this check is built into the `SdFeatureTree` dialog function. If a custom setup is selected and there is not enough space, the `SdFeatureTree` dialog function displays a warning message box. When the end user clicks OK on the message box, the feature selection dialog is still displayed.

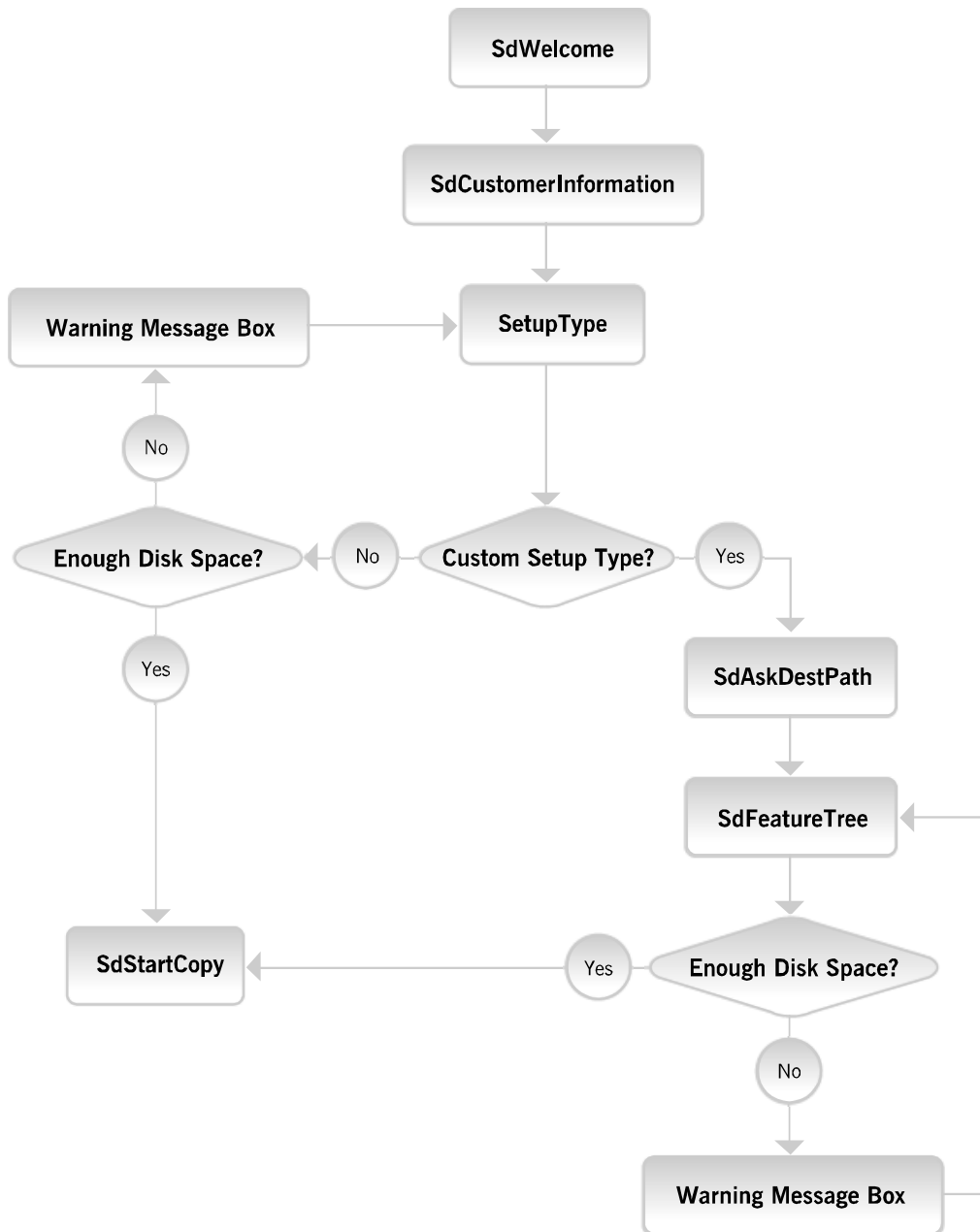


Figure 12-12: *The default logic of the user interface for a Standard project created in the IDE.*

The final dialog displayed before starting to make changes to the target system is displayed by the `SdStartCopy` dialog function. This dialog is used to display a list of all the selections that the end user made. This allows the end user to go back and change any of their selections. By default this dialog is empty since there is no code to add information to the `LIST` variable that is used to populate this dialog. If the end user reviews the selections and decides to go back and change something, the dialog that is launched when the Back button is clicked depends on whether a custom setup type was selected or not. If a custom setup was selected, the execution jumps to the `Dlg_SdFeatureTree: label`. Otherwise, execution of the code jumps to the `Dlg_SetupType: label`.

The final function call in the code is made to the `Enable` function, passing it the `STATUSEX` constant. This displays a progress dialog that shows the installation's progress. The `STATUSEX` constant forces the display of a progress dialog that is the same size as the dialogs used in the rest of the user interface. Using a different constant displays a smaller progress dialog.

Everything discussed here can be used to understand the default user interface that is implemented in the `OnFirstUIAfter`, `OnMaintUIBefore`, and `OnMaintUIAfter` event handlers. Now that you understand the default user interface for a fresh installation, we can take a look at an example of how to change it to suit your needs. A complete description of all available functions is found in the InstallScript Language Reference in the InstallShield Developer online help.

Modifying the Default User Interface

To get some experience working with the InstallScript dialog functions, you can modify the default user interface that is provided in the `OnFirstUIBefore` event handler. The changes that you will make in this example will change the flow of the dialogs that are displayed. This example will also add some additional functionality to the user interface so that the user is required to enter a valid serial number before they can continue. This serial number validation will be performed by a DLL that was used in Chapter 8.

The flow of the user interface as you are changing it in this example is shown in Figure 12-13.

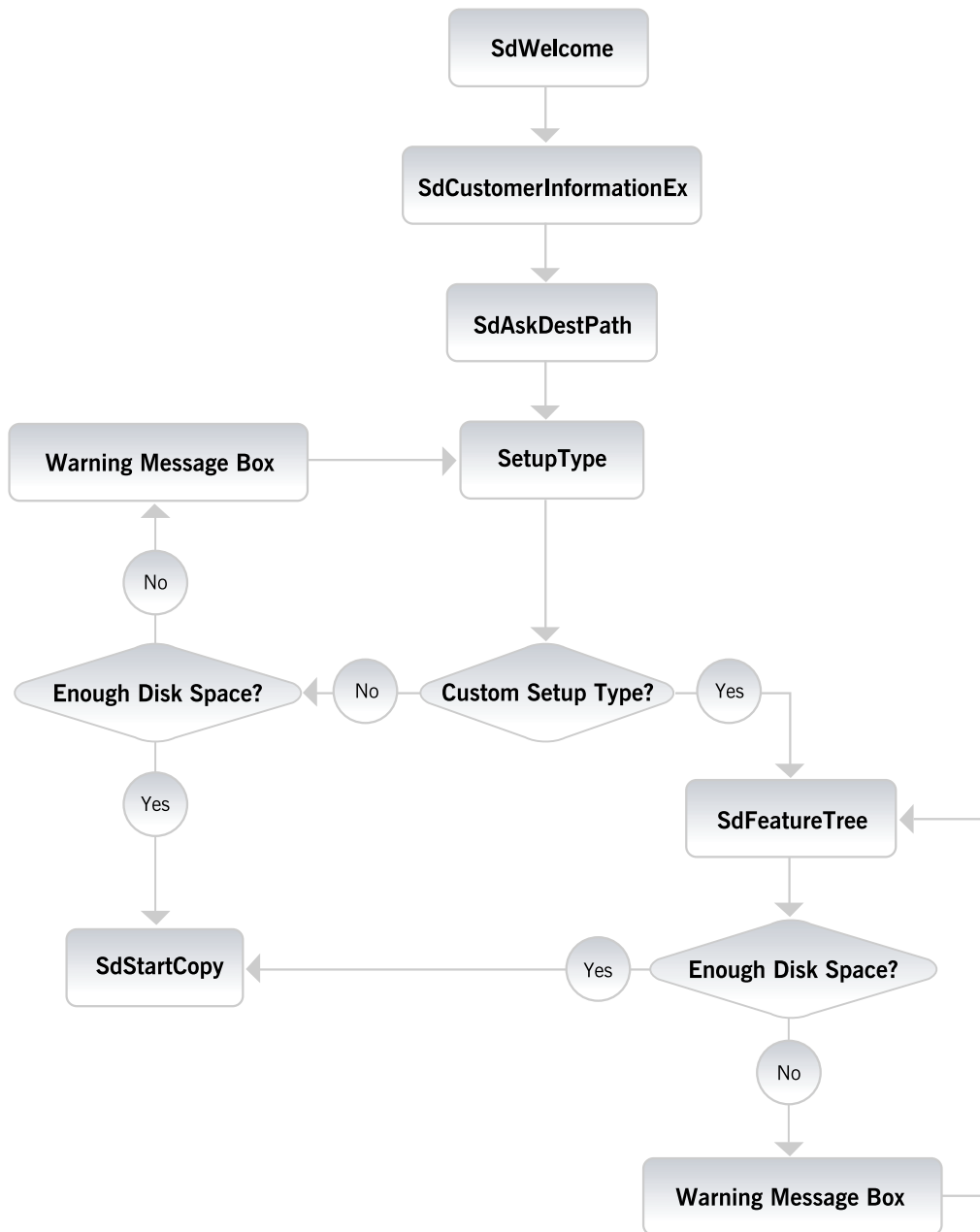


Figure 12-13: *Modified user interface logic for the Standard project Developer Art application.*

The example that we discuss here has you make the three changes to the default user interface as listed below:

1. Move the `SdAskDestPath` dialog before the `SetupType` dialog so, even if the end user does not want to perform a custom installation, they can still change the installation location.
2. The `SdCustomerInformation` dialog is to be changed out with the `SdCustomerInformationEx` dialog, which forces the end user to enter a serial number in order to proceed with the installation. You will use the DLL that you created in Chapter 8 to perform serial number validation.
3. Collect information about the options selected by the end user so the selections can be displayed in the `SdStartCopy` dialog.

The code to implement the changes listed above is provided in Figure 12-14. In this revised version of the `OnFirstUIBefore` event handler, you will borrow code from the serial number validation example in Chapter 8.

Other than rewriting and compiling the code shown in Figure 12-14, you need to add the `SerialNumber.dll` file to the Language Independent setup files in our project. Do this in the Setup Files/Billboards view under Step 5 in the View List.

```
#define MAX_TRYs    3
prototype stdcall LONG SerialNumber.ValidateSerialNo(BYVAL STRING,
                                                    BYVAL STRING, BYVAL STRING, BYVAL LONG, BYVAL LONG);
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
//  FUNCTION:    OnFirstUIBefore
//
//  EVENT:      FirstUIBefore event is sent when installation is run
//              for the first time on given machine. In the handler
//              installation usually displays UI allowing end user
//              to specify installation parameters. After this
//              function returns, ComponentTransferData is called
//              to perform file transfer.
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
function OnFirstUIBefore()
    NUMBER nResult, nSetupType, nvSize, nUser;
```

Figure 12-14: *The revised OnFirstUIBefore event handler.*


```

LONG    lStart, lIncrement, lValidate;
INT     iTryCnt;
STRING  szTitle, szMsg, szQuestion, svName, svCompany, szFile;
STRING  szLicenseFile, svSerial, szSetupType;
STRING  szProduct, szVersion, szDLLName;
LIST    list, listStartCopy;
BOOL    bCustom;
begin

    // Enable the background window.
    SetTitle( @PRODUCT_NAME, 24, WHITE );
    SetTitle( @PRODUCT_NAME, 0, BACKGROUNDCAPTION );
    Enable( FULLWINDOWMODE );
    Enable( BACKGROUND );
    SetColor(BACKGROUND, RGB (0, 128, 128));

    // Set the location of the DLL used
    // for validating the serial number.
    szDLLName = SUPPORTDIR ^ "SerialNumber.dll";
    UseDLL(szDLLName);

    // Initialize the arguments to be used in the
    // SdCustomerInformationEx function and the call to the DLL.
    szProduct = "ABCDEF";
    szVersion = "0750";
    lStart = 1000000519;
    lIncrement = 519;
    iTryCnt = 1;
    // Set the default setup type.
    nSetupType = TYPICAL;

Dlg_SdWelcome:

    szTitle = "";
    szMsg = "";
    nResult = SdWelcome(szTitle, szMsg);
    if (nResult = BACK) goto Dlg_SdWelcome;

    szTitle = "";
    svName = "";
    svCompany = "";

Dlg_SdCustomerInformation:

    // Null the serial number for every try.
    svSerial = "";

```

Figure 12-14: *Continued.*

```

// Let a certain number of attempts at entering a serial number.
if(iTryCnt > MAX_TRYs) then
    MessageBox("Too many attempts.\n" +
        "The installation will now terminate.", SEVERE);
    abort;
endif;

nResult = SdCustomerInformationEx(szTitle, svName,
                                svCompany, svSerial, nUser);
if (nResult = BACK) goto Dlg_SdWelcome;

// When the Next button is clicked validate the
// serial number that was entered.
if(nResult = NEXT) then
    lValidate = ValidateSerialNo(svSerial, szProduct,
                                svVersion, lStart, lIncrement);

    if(lValidate < 0) then
        iTryCnt++;
        goto Dlg_SdCustomerInformation;
    endif;
endif;

Dlg_SdAskDestPath:
nResult = SdAskDestPath(szTitle, szMsg, INSTALLDIR, 0);
if (nResult = BACK) goto Dlg_SdCustomerInformation;

Dlg_SetupType:
szTitle = "";
szMsg = "";
nResult = SetupType(szTitle, szMsg, "", nSetupType, 0);
if (nResult = BACK) then
    goto Dlg_SdAskDestPath;
else
    nSetupType = nResult;
    if (nSetupType != CUSTOM) then
        nvSize = 0;
        FeatureCompareSizeRequired(MEDIA, INSTALLDIR, nvSize);
        if (nvSize != 0) then
            MessageBox(szSdStr_NotEnoughSpace, WARNING);
            goto Dlg_SdAskDestPath;
        endif;
        bCustom = FALSE;
        goto Dlg_SdStartCopy;
    else
        bCustom = TRUE;
    endif;
endif;
endif;

```

Figure 12-14: *Continued.*

```

Dlg_SdFeatureTree:

    szTitle    = "";
    szMsg      = "";

    if (nSetupType = CUSTOM) then
        nResult = SdFeatureTree(szTitle, szMsg, INSTALLDIR, "", 2);
        if (nResult = BACK) goto Dlg_SetupType;
    endif;

Dlg_SdStartCopy:

    szTitle = "";
    szMsg   = "";

    // Add the selection information to the list
    // so that it can be displayed in the SdStartCopy dialog.
    listStartCopy = ListCreate( STRINGLIST );
    ListAddString(listStartCopy,
                  "User Name: " + svName, AFTER);
    ListAddString(listStartCopy, " ", AFTER);
    ListAddString(listStartCopy,
                  "Company Name: " + svCompany, AFTER);
    ListAddString(listStartCopy, " ", AFTER);
    ListAddString(listStartCopy,
                  "Serial Number: " + svSerial, AFTER);
    ListAddString(listStartCopy, " ", AFTER);
    ListAddString(listStartCopy, "Install Location:", AFTER);
    ListAddString(listStartCopy, " " + INSTALLDIR, AFTER);

    if(nSetupType = CUSTOM) then
        szSetupType = "Custom setup type selected.";
    elseif(nSetupType = TYPICAL) then
        szSetupType = "Typical setup type selected.";
    elseif(nSetupType = COMPACT) then
        szSetupType = "Compact setup type selected.";
    endif;

    ListAddString(listStartCopy, " ", AFTER);
    ListAddString(listStartCopy, szSetupType, AFTER);

    nResult = SdStartCopy( szTitle, szMsg, listStartCopy );

    ListDestroy(listStartCopy);

```

Figure 12-14: *Continued.*

```

    if (nResult = BACK) then
        if (!bCustom) then
            goto Dlg_SetupType;
        else
            goto Dlg_SdFeatureTree;
        endif;
    endif;

    // setup default status
    Enable(STATUSEX);

    // Free the DLL from memory.
    UnUseDLL("SerialNumber.dll");

    return 0;
end;

```

Figure 12-14: *Continued.*

The first thing you do to create the script shown in Figure 12-14 is to cut the call to the `SdAskDestPath` function, including the `Dlg_SdAskDestPath:` label, and paste it back into the script just before the `Dlg_SetupType:` label. Fix the response to the end user clicking the Back button for the `SdAskDestPath`, `SetupType`, and `SdFeatureTree` dialogs. The response to clicking the Back button has to recognize the new forward flow of the user interface.

The next step is to replace the call to the `SdCustomerInformation` function with a call to the `SdCustomerInformationEx` function. This requires you to declare a new `STRING` variable to hold the serial number entry. Along with this call to `SdCustomerInformationEx`, you need to add some code to be able to use the `ValidateSerialNo` function exported from `SerialNumber.dll`. The serial number validation is performed when the code detects that the Next button is clicked. If the entry does not validate, the user is placed back into the `SdCustomerInformationEx` dialog with a `NULL` value for the serial number. A count is kept of the number of tries. After three tries, a message box is displayed and the installation aborts.

The final change that is made to the original script is to collect information about the various selections that have been made in the installation. This information is then placed into a string list, which is displayed in the `SdStartCopy` dialog. In this example, the user name, company name, serial number, installation location, and setup type are collected.

Creating a Custom Dialog Box

This section introduces how to create a custom dialog box in a Standard project. Because there are many issues involved in creating dialogs, this section covers only the basics. The first thing that you will do is to create a dialog template with a dialog function that provides some basic functionality. You will create your own dialog template even though there is a template provided by InstallShield Developer. Creating this template on your own will teach a lot of the basics of creating custom dialogs using InstallScript. You will then use this template to create a more robust dialog that can be used in an installation.

Creating the BasicDialog Dialog

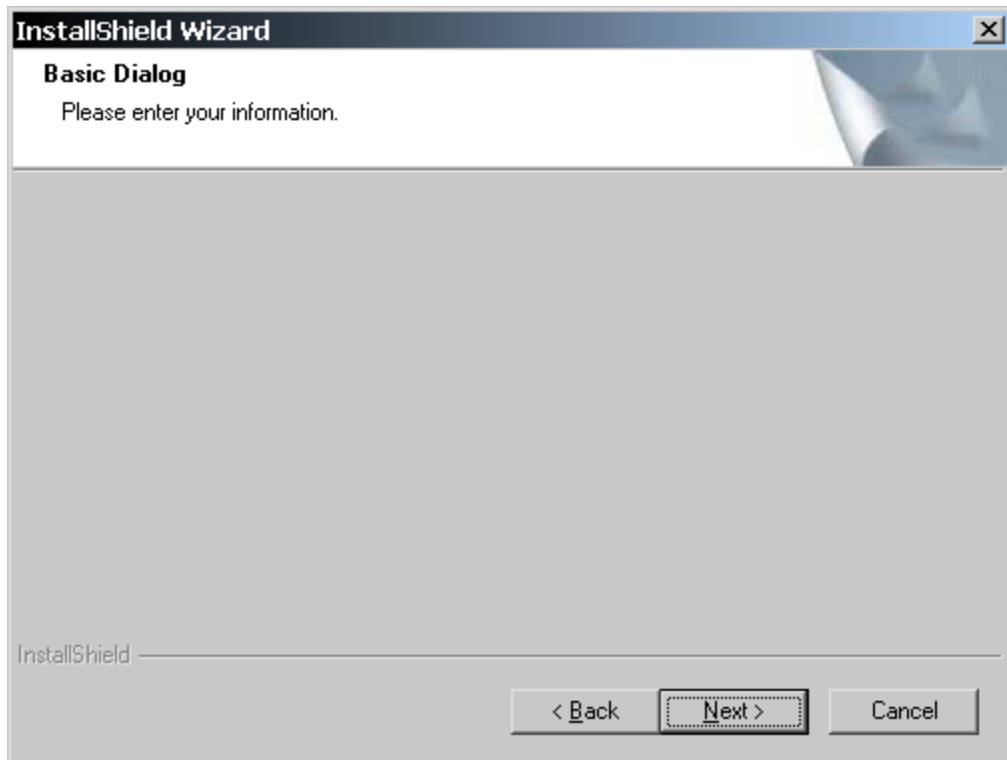


Figure 12-15: *The design of the BasicDialog dialog.*

Creating a simple template dialog provides a foundation for working with custom dialogs without getting into too much detail. Our `BasicDialog` dialog will have the three standard buttons, Next, Back, and Cancel, as well as a banner bitmap and some static graphics controls so it looks like the other dialogs. You will also have a dialog and sub-title that you can customize when you call the dialog function. When finished, this template dialog will look like what is shown in Figure 12-15.

To create the `BasicDialog` dialog, you will perform the following series of steps.

1. Set up an environment in which to create your custom dialogs.
2. Use the Dialog Editor to create the resource-only dynamic link library that contains the dialog template.
3. After the dialog template is defined, create the dialog function that will run the dialog.
4. Test the dialog template by calling the dialog function from within the `OnFirstUIBefore` function.

These steps are discussed in the following sections.

SETTING UP THE CUSTOM DIALOG ENVIRONMENT

To set up an efficient environment, you need to create a directory structure under `C:\MySetups` and collect bitmaps and icons that can be used in the created dialog. The directory structure is shown in Figure 12-16. To create this environment perform the following steps:

1. Create a folder named `Dialogs` under the `MySetups` folder.
2. Under the `Dialogs` folder, create a folder named `Custom Dialogs`. Also create another folder called `Graphics`.
3. Under the `Custom Dialogs` folder, create folders for the dialog functions' source code. These folders are named `Include`, `Lib`, and `Src` respectively.

- Under the Graphics folder create folders to hold bitmaps and icons. These folders are named Bitmaps and Icons respectively.

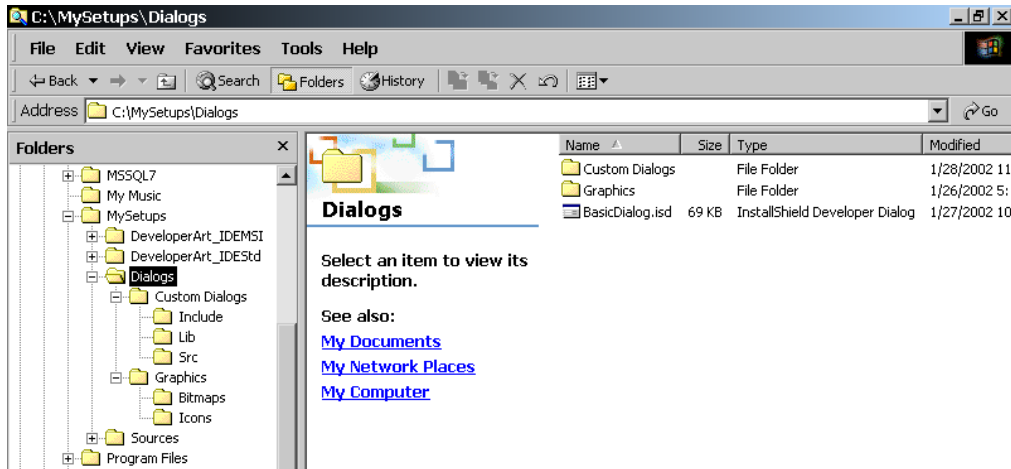


Figure 12-16: *Custom dialog directory structure.*

The Include folder under Custom Dialogs is where you will place the header files that contain the dialog function prototypes and the definition of the constants that you will use in the dialog functions. The Lib folder is where you can place the script library that will contain all the dialog functions for all the custom dialogs that you want to create. The Src folder is where you will place the InstallScript .rul files that contain the definition of the dialog functions. Directly under the Dialogs folder you will place any dialogs that you export from your projects into .isd files.

The examples in this book use only one bitmap. This bitmap is in a file named IsDialogBanner.ibd in the following location:

```
C:\Program Files\InstallShield\Developer\Redist\Language Independent
    \OS Independent
```

You need to copy this file over to the Graphics\Bitmaps folder and change the extension to .bmp. A copy of this bitmap can also be found on the CD-ROM at the back of the book. Once the directory structure is complete and the IsDialogBanner.bmp file has been created, you can move on to step 2 in the process. Eventually, you will copy the contents of the Dialogs folder to the following location:

%USERPROFILE%\My Documents\MySetups\Dialogs

It is in this location that the Dialog Gallery in InstallShield Developer looks for any exported dialogs so that you can use them as the starting point for any new custom dialog.

CREATING THE DIALOG TEMPLATE

To begin the creation of the dialog template do the following:

1. Go to the Dialogs view under Step 4. Right-click on the All Dialogs icon in this view and select New Dialog.
2. Name the new dialog that is created "BasicDialog."
3. Click on the English (United States) icon under the BasicDialog you have just created to launch the Dialog Editor (Figure 12-17). You now have a blank form on which you can place the controls that make up the BasicDialog dialog.

The Dialog Editor contains two new toolbars. The toolbar just above the blank form is the Controls toolbar that gives a selection of controls that you can place on your dialog. Place your cursor over any of the controls on this toolbar to see a tooltip that displays the type of control represented by the button.

To place one of these controls on a dialog box form do the following:

1. Click the desired control.
2. Move the cursor to the form.
3. Hold down the left mouse button and draw the control onto the form.

The controls that can be placed on a dialog were discussed earlier in this chapter. The toolbar just above the Controls toolbar is the Layout toolbar. When two or more controls are selected on a dialog form, the buttons on this toolbar can be used to organize the selected controls. Place the mouse cursor on top of any button to see a tooltip that displays the button's purpose.

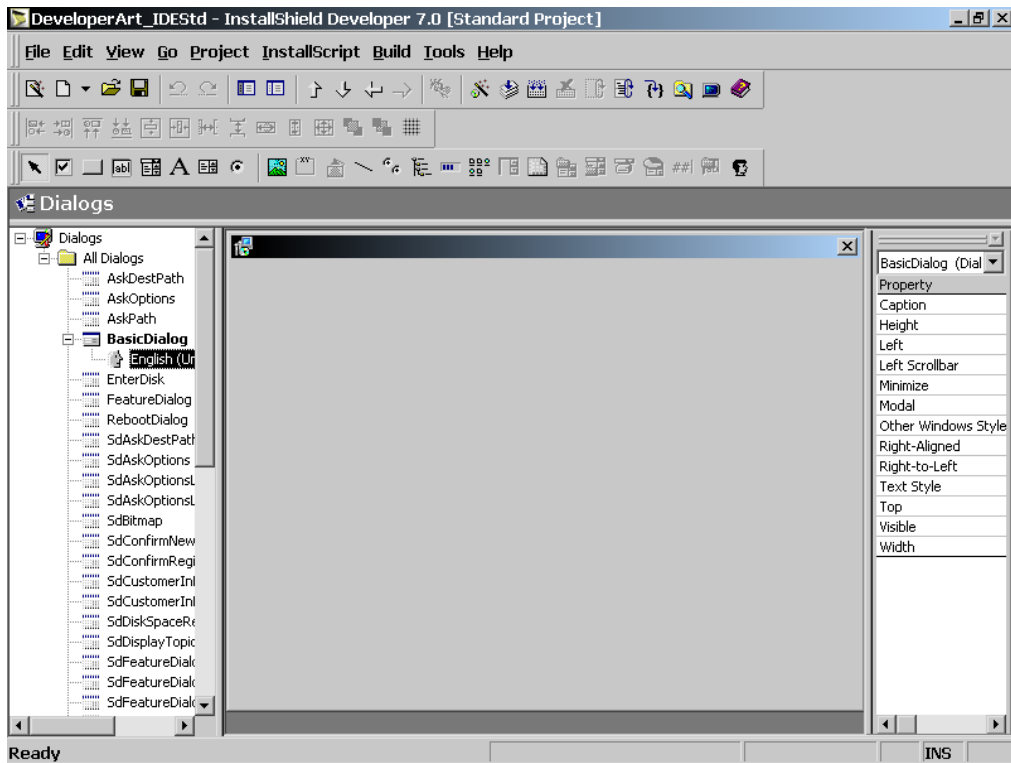


Figure 12-17: *The Dialog Editor showing a blank dialog form.*

On the right of the Dialog Editor is a property sheet that contains a field for the names of the properties and a field for the property values. Every control that you place on the dialog form has a row in this property sheet. Before placing controls on the dialog form, you should set the properties for the dialog. For this example, you can leave the default values for all the properties, with the exception of the Caption property.

The Caption property contains a string that appears in the dialog's title bar. To make this dialog look like all the other InstallShield Developer dialogs, use "InstallShield Wizard" here. When you enter this string, you need to assign it a unique string ID. A unique string ID allows you to localize this dialog and it avoids a potential conflict with other strings IDs in any project to which you might import this dialog.

For example, suppose you create your BasicDialog dialog and accept the default string IDs provided by InstallShield Developer. Now assume that you export this

dialog for use in other projects. Now, if one of the default string IDs for a string in this dialog is ID_STRING35 and this same string ID is being used to identify a string being used as the description of a feature you will get a conflict. This conflict, if not resolved, can cause one of the strings to be used in the dialog as well as for the description of the feature.

When you import the dialog into this project, you are presented with two choices, use the string as assigned to this non-unique string ID as defined in the project or use the string as defined in the dialog file. You can avoid a conflict situation by using unique string IDs for all strings in the dialogs that you create.

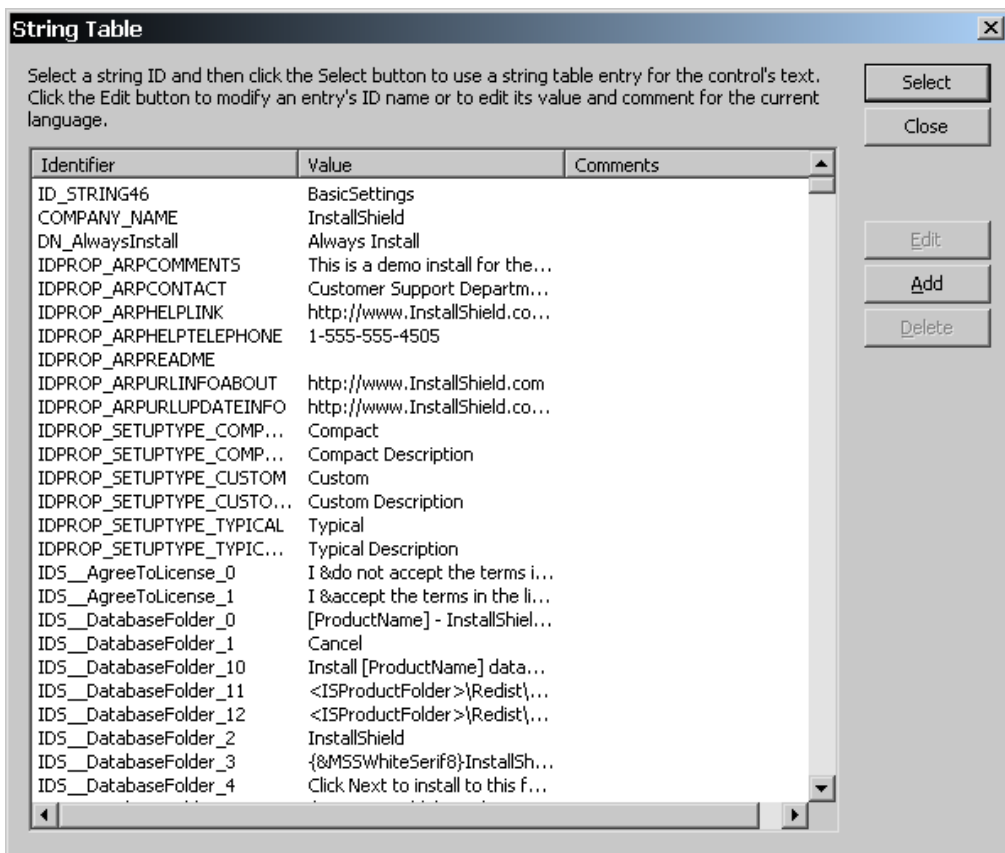


Figure 12-18: *The String Table dialog as launched from the Dialog Editor.*

To enter the caption to be used in your BasicDialog dialog do the following:

1. Click in the value field for the Caption property then click the ellipsis button on the right side of the field. This launches the String Table dialog, which provides the ability to create a new string and assign it to a unique string ID (Figure 12-18).
2. Click the Add button to launch the String Entry dialog (Figure 12-19). Then type what is shown in Figure 12-19 in the ID and Text fields. Click OK to add the string to the String Table and return to the String Table dialog.

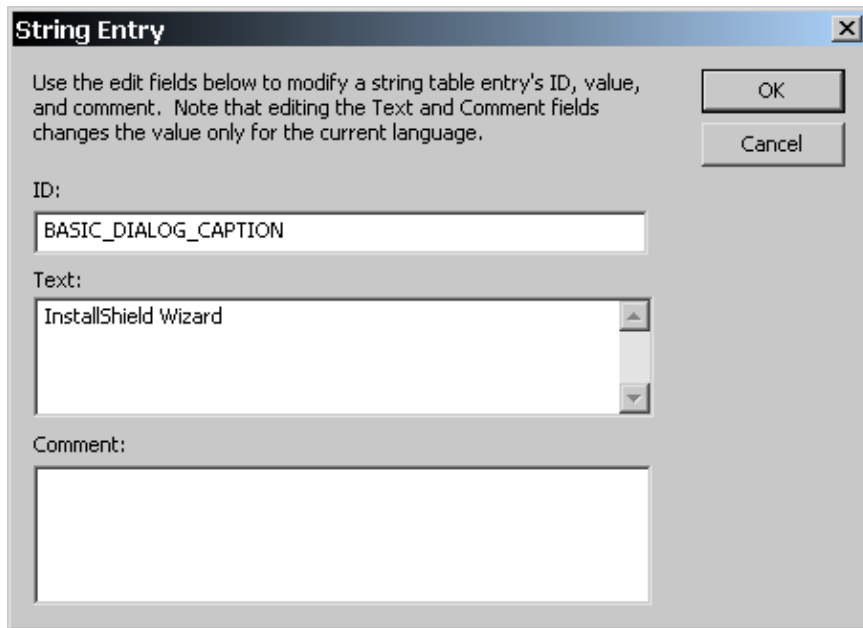


Figure 12-19: *The String Entry dialog.*

3. Find the BASIC_DIALOG_CAPTION string ID, highlight it, and then click Select. This selects the string for the Caption property value.

You will use this operation for all the strings that you generate for the BasicDialog dialog. Note in Figure 12-19 the format used for creating the unique string ID. You will use this same format for all the other string IDs that you create. The first part of the string ID will be BASIC_DIALOG_ and the last part will be the name of the property or the name of the control on which you are placing the text.

Now click in the value field of the Other Windows Styles property and then click the ellipsis button to launch the Other Window Styles dialog (Figure 12-20). This dialog provides the option to add or remove Windows styles from the resource that will be created for the dialog box when you build the project.

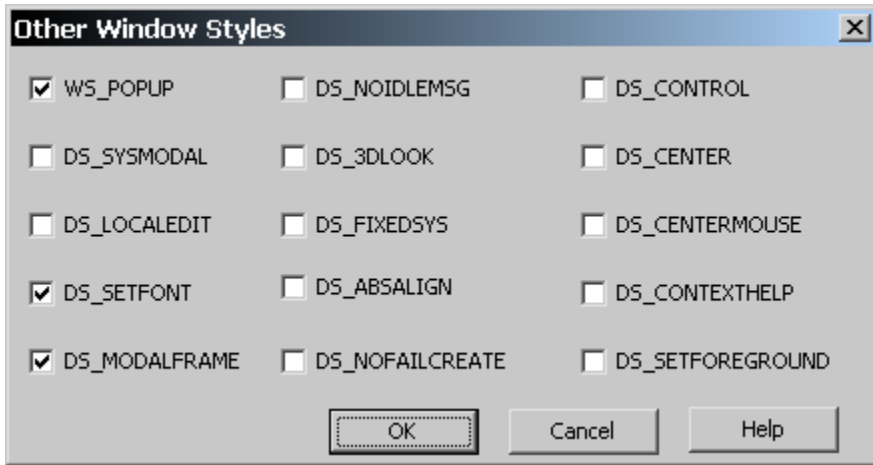


Figure 12-20: *The Other Windows Styles dialog box for a dialog resource.*

Using the Other Windows Styles dialog box allows you to add additional styles and to remove styles from a dialog box resource. For the BasicDialog dialog, you do not need to change any of the default styles that are selected. The default styles define that this dialog will be a modal, popup window, and that the resource header will contain the FONT statement that specifies the font to be used to write all text in the dialog. By default all dialog box resources that are created in the Dialog Editor have the WS_CAPTION and WS_SYSMENU styles. This means that all of your dialogs will have a title bar and that there will also be a system menu.

This chapter discusses how to add controls to the dialog, according to the type of control being added. We will first discuss how to add simple graphic static controls, and then how to add text static controls, and push button controls. First, you will add a bitmap and two rectangles using the Line control from the Controls toolbar.

You need to reserve the identifiers of 1, 2, 7, 9, 12, 50, 51, and 52 for use with the other controls that you will place on the dialog. The control identifiers 1, 2, and 12 are used for the Next, Cancel, and Back buttons respectively. The control identifiers 7, 9, and 52 are used for special purposes and the control identifiers 50 and 51 are used for

the two text static controls that are placed on the banner bitmap and serve as the dialog title and sub-title respectively. If you add a control and the default identifier is one of these reserved numbers, you need to change it to something else. Table 12-1 shows those properties that you need to set in the Dialog Editor. The other properties for these simple graphic static controls are left as the default values with the exception of the Control Identifier property.

There are two properties that are used to set the position of the control in the dialog box: `Top` and `Left`. The `Left` property is the distance to the left side of the control from the left edge of the dialog box. The `Top` property is measured from the top of the dialog box, which is coincident with the bottom edge of the title bar of the dialog box.

Table 12-1: Simple Graphic Static Control Property Values

Control Type	Property	Value
Bitmap	Name	BannerBitmap
	File Name	<DialogBitmaps>\IsDialogBanner.bmp
	Height	44
	Left	0
	Other Windows Styles	None
	Top	0
	Width	374
Line	Name	BitmapLine
	Height	2
	Left	0

Table 12-1: Simple Graphic Static Control Property Values (Continued)

Control Type	Property	Value
Line	Other Windows Styles	SS_ETCHEDHORZ and WS_GROUP
	Top	44
	Width	374
Line	Name	BrandingLine
	Height	2
	Left	48
	Other Windows Styles	SS_ETCHEDHORZ and WS_GROUP
	Top	227
	Width	325

As shown in Table 12-1, a path variable is used to identify the folder in which the dialog bitmaps are stored. When you browse to the location for the bitmap file, the Path Variable dialog appears. Enter the path variable for this location. The control names are intuitive, indicating the purpose of the control. This becomes important when you are trying to distinguish a number of controls of the same type, which have different purposes. The BitmapLine control is used to give a three-dimensional appearance to the banner bitmap at the top of the dialog. The BrandingLine control is used to brand the dialog, along with the word InstallShield that is added automatically to the resource file when the project is built.

Next, you will add two text static controls to your dialog box. These are the controls used for the dialog's title and sub-title. The text in these two controls is displayed on top of the banner bitmap. You can configure what goes into these two text fields or

have default strings displayed. The dialog function sets the text in these two controls if you want something different than the default string that you provide when you create the control. The values of the properties that you need to set are shown in Table 12-2. For the other properties, leave the default value making sure that the Enabled property is set to True.

Table 12-2: Text Static Control Property Values

Control Type	Property	Value
Text Area	Name	Title
	Control Identifier	50
	Height	12
	Left	12
	Other Windows Styles	None
	Text	Basic Dialog
	Top	4
	Transparent	True
	Width	248
Text Area	Name	SubTitle
	Control Identifier	51
	Height	24
	Left	19
	Other Windows Styles	None

Table 12-2: Text Static Control Property Values (Continued)

Control Type	Property	Value
Text Area	Text	Please enter your information.
	Top	19
	Transparent	True
	Width	240

As mentioned, the control IDs for the two text static controls need to be 50 and 51, respectively. The value that you entered for the Text property is the default string that is displayed in the text control, unless the dialog function inserts a different string. Because these text controls are being displayed on top of a bitmap, you need to set the value of the Transparent property to True. This allows the color of the bitmap to show through the text control.

The last three controls that you need to add to the dialog are the Next, Back, and Cancel buttons. The values you need to enter for the non-default properties for these three controls are shown in Table 12-3.

Table 12-3: Push Button Control Property Values

Control Type	Property	Value
Push Button	Name	Next
	Control Identifier	1
	Default	True
	Height	17
	Left	242

Table 12-3: Push Button Control Property Values (Continued)

Control Type	Property	Value
Push Button	Other Windows Styles	None
	Tab Index	0
	Text	&Next >
	Top	239
	Width	57
	Name	Back
	Control Identifier	12
	Height	17
	Left	187
	Other Windows Styles	None
Push Button	Tab Index	1
	Text	< &Back
	Top	239
	Width	57
	Name	Cancel
	Control Identifier	2
	Cancel	True

Table 12-3: Push Button Control Property Values (Continued)

Control Type	Property	Value
	Height	17
	Left	307
	Other Windows Styles	None
	Tab Index	2
	Text	Cancel
	Top	239
	Width	57

As indicated in Table 12-3, you need to provide a value to the Control Identifier property of 1 for the Next button, 12 for the Back button, and 2 for the Cancel button. The control identifiers for the Next and Back button are used as the return values from the dialog function when the user clicks one of these buttons.

For the Next button, set the Default property to True, which makes this button the control that responds when the end user presses the Enter key. For the Cancel button, set the Cancel property to True so the system Close button performs the same operation as assigned to the Cancel button.

For this BasicDialog dialog, set the Tab Stop property to 0, 1, and 2 for the Next, Back, and Cancel buttons respectively. This allows the end user to tab between the buttons in order.

Before you can use this dialog, you have to create a dialog function. Before you create the dialog function, however, you can build the project and review what is created. After the build completes, go to the following location to see what was created. You now have a custom dialog in your project.

C:\MySetups\DeveloperArt_IDEStd

In this location, you will see three new files with the names `_ISUser1033.RC`, `_ISUser1033.RES`, and `_ISUser1033.DLL`. If you open the `.rc` file in Notepad, you should see what is shown in Figure 12-21.

```

#include <windows.h>
#include <commctrl.h>

#ifdef WIN32
LANGUAGE 0x09, SUBLANG_DEFAULT
#pragma code_page(1252)
#endif

BasicDialog DIALOGEX 29127,26886,332,218
STYLE DS_SETFONT | DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSTEMMENU
CAPTION "InstallShield Wizard"

FONT 8,"MS Sans Serif",0,0,0x1
BEGIN
    DEFPUSHBUTTON    "<Next >",1,215,196,50,14
    PUSHBUTTON       "< <Back",12,166,196,50,14
    PUSHBUTTON       "Cancel",2,272,196,50,14
    LTEXT            "Basic Dialog",50,10,3,220,9,,NOT WS_GROUP,WS_EX_TRANSPARENT
    CONTROL          "",15,"Static",SS_ETCHEDHORZ | WS_DISABLED | WS_GROUP,42,186,288,1
    CONTROL          "",16,"Static",SS_ETCHEDHORZ | WS_DISABLED | WS_GROUP,0,36,332,1
    CONTROL          10001,11,"Static",SS_BITMAP | WS_DISABLED,0,0,332,36
    LTEXT            "Please enter your information.",51,16,15,213,19,,NOT WS_GROUP,WS_EX_TRANSPARENT
    LTEXT            "",7,2,182,40,10,NOT WS_VISIBLE
END

////////////////////////////////////
//
// Bitmaps
//
10001          BITMAP  "C:\\MySetups\\Dialogs\\Graphics\\Bitmaps\\IsDialogBanner.bmp"

```

Figure 12-21: *The resource file for the BasicDialog dialog as created during the build.*

The file `_ISUser1033.DLL` is streamed into the Binary table during the build and, when the installation is run, it is streamed out with the name `_ISUser.dll`. It is this file that the dialog function accesses in order to display the dialog on the screen. The creation of the dialog function is the next topic.

CREATING THE DIALOG FUNCTION

This task is to create an InstallScript function with the same name as the name of the dialog box that you can call from within `Setup.rul`. The function name does not need to be the same as the dialog name, but it is a good practice to follow to avoid

confusion. The parameters to the dialog function need to be variables that are used for passing information to the dialog or variables that are used to retrieve information from the dialog.

We are going to keep things simple with this first dialog function. The dialog function displays the dialog template and to responds appropriately to the end user clicking on any of the three buttons on the dialog. The BasicDialog function that runs the dialog template that you have just created is shown in Figure 12-22.

```

////////////////////////////////////
//
// File Name:      BasicDialog.rul
//
// Description:    InstallScript file for a basic custom dialog
//                 that can be used as a template.
//
// Comments:      This script demonstrates the creation
//                 of an InstallScript custom dialog.
//
////////////////////////////////////

// Include Windows API prototypes that are not normally used.
#include "winapi.h"

// Defines for commonly used controls
#define CUST_BUTTON_NEXT      1
#define CUST_BUTTON_CANCEL    2
#define CUST_BUTTON_BACK     12
#define CUST_TEXT_TITLE       50
#define CUST_TEXT_SUBTITLE    51

prototype INT BasicDialog(BYVAL STRING, BYVAL STRING);
////////////////////////////////////
// Function:      BasicDialog
//
// Purpose:       This function provides the skeleton that can be
//                 used to create dialog functions for more complex
//                 dialogs.
//
////////////////////////////////////
function INT BasicDialog(szTitle, szSubTitle)
STRING  szDlg;
INT     nId;
HWND    hwndDlg, hwndTitle, hwndSubTitle;
BOOL    bDone;
begin

```

Figure 12-22: *The Dialog function for the BasicDialog dialog.*

```

// Define the name for the dialog to be used
// in the custom dialog functions.
szDlg = "BasicDialog";

// Check the global variable to see if InstallScript
// has already been initialized for custom dialogs.
// Initialize the environment if not already initialized.
if(!bSdInit) then
    SdInit();
endif;

// Define the dialog using the string name that is
// used to identify it in the resource file.
if(EzDefineDialog(szDlg, ISUSER, szDlg, 0) < 0) then
    MessageBox("Error defining dialog box", INFORMATION);
    abort;
endif;

// Initialize the message loop control variable.
bDone = FALSE;

// Loop in the dialog until the user takes action
// to close the dialog by clicking on a button.
while(!bDone)

    // Display the dialog and retrieve messages
    // based on the users interaction with the dialog.
    nId = WaitOnDialog(szDlg);

    switch(nId)

        // The first message sent before the dialog
        // is displayed. This is where we can
        // initialize controls in the dialog.
        case DLG_INIT:
            hwndDlg = CmdGetHwndDlg(szDlg);
            // Set the custom title of the dialog.
            if(szTitle != "") then
                hwndTitle = GetDlgItem(hwndDlg, CUST_TEXT_TITLE);
                SetWindowText(hwndTitle, szTitle);
            endif;

            // Set the custom sub-title of the dialog.
            if(szSubTitle != "") then
                hwndSubTitle = GetDlgItem(hwndDlg,
                                           CUST_TEXT_SUBTITLE);
                SetWindowText(hwndSubTitle, szSubTitle);
            endif;

```

Figure 12-22: *Continued.*

```

        // The code in this case statement responds to
        // the end user clicking the Next button.
        case CUST_BUTTON_NEXT:
            nId = CUST_BUTTON_NEXT;
            bDone = TRUE;

        // The code in this case statement responds to
        // the end user clicking the Cancel button or the
        // Close button.
        case CUST_BUTTON_CANCEL:
            Do(EXIT);

        // The code in this case statement responds to
        // the end user clicking the Back button.
        case CUST_BUTTON_BACK:
            nId = CUST_BUTTON_BACK;
            bDone = TRUE;

        endswitch;

    endwhile;

    // Close the dialog.
    EndDialog(szDlg);

    // Release memory used by the dialog.
    ReleaseDialog(szDlg);

    // Return the control ID of
    // the control that was clicked.
    return nId;

end;

```

Figure 12-22: *Continued.*

As shown in the code (Figure 12-22), the first thing you do is to define five constants that you can use in all custom dialogs that you create. You could use the constants that are used for the built-in script dialogs but for this example we are creating our own constants. Following the definition of the constants, the `BasicDialog` function is prototyped. This function takes two strings as arguments with the first argument being the dialog title and the second argument being the dialog sub-title.

The first operation is to initialize a string variable to the name of the dialog that this function is creating. You will use this string variable in any function that requires a reference to the dialog. The second operation is to check if the environment has already been initialized to run a script dialog box. Initialization consists of two

operations, the loading of strings from the string resources in `_ISUser.dll` and `_isres.dll` and the setting of the check boxes to have the Windows 95 style. If you create `_ISUser.dll` with the Dialog Editor in InstallShield Developer, there will be no string resource since that is not one of the available options. Once initialization is performed, it does not have to be done again for any other script dialog box function. The `bSdInit` global variable is used to indicate whether initialization has occurred or not.

After making sure that the environment is initialized, the code calls the `EzDefineDialog` function, which is a wrapper around the `DefineDialog` built-in function with some of the arguments defined. The main purpose of this function is to load the dialog template into memory. The reference to this location is held in the `szDlg` variable. You pass to the `EzDefineDialog` function the name of the dialog, the name and location of the resource-only DLL in which the dialog template is defined, and the identifier of the particular dialog template.

When you create dialog boxes using the Dialog Editor in InstallShield Developer, the identifier of the dialog template is the name you give to the dialog. If you use this name to set the value of the `szDlg` variable, you can use this variable as the argument passed for the `szDialogName` and `szDialogID` parameters to the `EzDefineDialog` function as is done in the code shown in Figure 12-21. The `EzDefineDialog` function does not actually display the dialog on the screen. That is the purpose of the `WaitOnDialog` function.

The next part of the dialog function is the message loop. This is where the code enters a while loop waiting for the end user to take action in the dialog. This loop continues until the `bDone` variable is set to `TRUE`. During this time, the `WaitOnDialog` function receives messages from Windows and passes them to your dialog function. Except for the first return value from the `WaitOnDialog` function, the values are the control IDs with which the end user has interacted. The first return value is equal to the `DLG_INIT` constant and this gives you the opportunity to initialize controls in the dialog before the dialog is displayed.

In the `BasicDialog` function, you set the strings to be shown in the `Title` and `SubTitle` text static controls. This is necessary only if the values passed to the `BasicDialog` function for these two string parameters are not `NULL`. If you pass custom strings to the `BasicDialog` function, you first obtain a handle to the control and then use the `SetWindowText` Windows API to set the string displayed in the

text static control to be the string passed to the function. To use the Windows API, you need to include the `WINAPI.H` header file at the top of your script. If there are no strings passed to the `BasicDialog` function, then you do nothing and the default strings you entered as the value of the `Text` property are displayed. After your code has handled the `DLG_INIT` message, the `WaitOnDialog` function calls the `ShowWindow` Windows API to display the dialog.

After the dialog is displayed, the code handles the actions that the end user performs in the dialog box. If the dialog has edit controls or check boxes, your code needs to capture any actions that indicate input from the user. When the end user clicks one of the three standard buttons, the message loop needs to take action to dismiss the dialog. If the `Cancel` button is clicked, action needs to be taken to terminate the installation. To do this in the `BasicDialog` function, make a call to the `DoInstallScript` function with the `EXIT` argument. This calls the standard exit message box if no `EXIT` handler is defined in your installation script.

After the while loop is finished, you destroy the dialog window and free the memory that was used by the dialog. First calling the `EndDialog` function and then calling the `ReleaseDialog` function accomplishes this.

The dialog function we have just looked at provides the bare essentials of how to create this type of function. In the next section, you will learn how to test this custom dialog.

TESTING THE BASICDIALOG DIALOG

The first thing that you should do is to place the statements that define the constants that you are using in a header file. You should also create another header file to hold the dialog function prototype. You can continue to add definitions and prototypes to these files as you create more custom dialogs. You can also have a third header file that includes both of the other files and the `winapi.h` header file. Good names for these three files are `MyCustDlgDefs.h`, `MyCustDlgFuncs.h`, and `MyCustDlg.h`. The content of these three header files is shown in Figure 12-23.


```

/////////////////////////////////////////////////////////////////
//
// File Name:      MyCustDlgDefs.h
//
// Description:    Header file for the custom InstallScript
//                dialog function defines.
//
// Comments:      This script demonstrates the creation
//                of a custom dialog.
//
/////////////////////////////////////////////////////////////////

// Defines for commonly used controls
#define CUST_BUTTON_NEXT      1
#define CUST_BUTTON_CANCEL    2
#define CUST_BUTTON_BACK     12
#define CUST_TEXT_TITLE       50
#define CUST_TEXT_SUBTITLE    51

/////////////////////////////////////////////////////////////////
//
// File Name:      MyCustDlgFuncs.h
//
// Description:    Header file for the custom InstallScript
//                dialog function prototypes.
//
// Comments:      This script demonstrates the creation
//                of a custom dialog.
//
/////////////////////////////////////////////////////////////////

prototype INT BasicDialog(BYVAL STRING, BYVAL STRING);
/////////////////////////////////////////////////////////////////
//
// File Name:      CustDlg.h
//
// Description:    Header file for the custom InstallScript dialogs.
//
// Comments:      This script demonstrates the creation
//                of a custom dialog.
//
/////////////////////////////////////////////////////////////////

#include "winapi.h"
#include "MyCustDlgDefs.h"
#include "MyCustDlgFuncs.h"

```

Figure 12-23: *Custom dialog header files.*

You can place these header files in the folder that you created at the beginning of this example. This location is as follows:

```
C:\MySetups\Dialogs\Custom Dialogs\Include
```

You should also create a separate .rul file for the body of the BasicDialog function. Call this file BasicDialog.rul. Place this file in the following folder:

```
C:\MySetups\Dialogs\Custom Dialogs\Src
```

So that the InstallShield compiler will be able to find the locations you have created for the include files and the .rul files, you need to modify the Compile Folders.ini file found in the following location:

```
C:\Program Files\InstallShield\Developer\Support
```

```
[Folders]
Folder0=<ISProductFolder>\Script\ISWi\Include
Folder1=<ISProductFolder>\Script\ISRT\Include
Folder2=<ISProductFolder>\Script\IFX\Include
Folder3=<ISProductFolder>\Script\Include
Folder4=C:\MySetups\Dialogs\Custom Dialogs\Include
Folder5=C:\MySetups\Dialogs\Custom Dialogs\Src

[Libraries]
```

Figure 12-24: *The Compile Folders.ini file with modifications for custom dialogs.*

You made modifications to this file in Chapter 8 when you created a script library. The changes required for this file are to add entries for Folder4 and Folder5 as shown in Figure 12-24.

To use your custom dialog in an installation, you need to include the MyCustDlg.h header file right after the include statement for the ifx.h header file. This will look as follows:

```
// Include header files ////////////////////////////////////////
#include "ifx.h"

// Include the custom dialog header file
#include "MyCustDlg.h"
```

At the bottom of the Setup.rul file, include the BasicDialog.rul file. This looks like the following:

```
#include "BasicDialog.rul"
```

You do not have to qualify the file name since you have identified the location of this file in the Compile Folders.ini file. Now you need to insert a call to the dialog function in the OnFirstUIBefore event handler. A good place to insert this dialog is between the CustomerInformation dialog and the SetupType dialog.

```
Dlg_SdWelcome:
    szTitle = "";
    szMsg = "";
    nResult = SdWelcome(szTitle, szMsg);
    if (nResult = BACK) goto Dlg_SdWelcome;

    szTitle = "";
    svName = "";
    svCompany = "";

Dlg_SdCustomerInformation:
    nResult = SdCustomerInformation(szTitle, svName,
                                   svCompany, nUser);
    if (nResult = BACK) goto Dlg_SdWelcome;
Dlg_BasicDialog:
    szTitle = "Basic Dialog Test";
    szMsg = "Click on any button to see what happens";
    nResult = BasicDialog(szTitle, szMsg);
    if(nResult = BACK) goto Dlg_SdCustomerInformation;

Dlg_SetupType:
    szTitle = "";
    szMsg = "";
    nResult = SetupType(szTitle, szMsg, "", nSetupType, 0);
    if (nResult = BACK) then goto Dlg_BasicDialog;
    else
        nSetupType = nResult;
        if (nSetupType != CUSTOM) then
            nvSize = 0;
            FeatureCompareSizeRequired(MEDIA, INSTALLDIR, nvSize);
            if (nvSize != 0) then
                MessageBox(szSdStr_NotEnoughSpace, WARNING);
                goto Dlg_SetupType;
            endif;
            bCustom = FALSE;
            goto Dlg_SdStartCopy;
        else
            bCustom = TRUE;
        endif;
    endif;
```

Figure 12-25: *Inserting the BasicDialog dialog into the user interface sequence.*

The section of code in the `OnFirstUIBefore` event handler to do this is shown in Figure 12-25. To insert the `BasicDialog` dialog do the following:

1. Define a new label called `Dlg_BasicDialog`. After this label, define values for the `szTitle` and the `szSubTitle` parameters to the `BasicDialog` function.
2. When you call the `BasicDialog` function, check to see if the return value is equal to the `BACK` system constant. The value of this system constant is 12, which is what the `BasicDialog` function returns when the Back button is pressed.
3. Have the `SetupType` dialog jump back to the `BasicDialog` dialog when the Back button is pressed.

With the changes discussed in this section, you can build the project and run the installation. You should see that the `BasicDialog` dialog appears between the `CustomerInformation` dialog and the `SetupType` dialog. You should also test that you can move back and forth through the user interface using the Next and Back buttons.

Creating a Dialog From a Template

You can now use the `BasicDialog` dialog to create a dialog that has controls on it and with which the end user can interact by entering information. The dialog that you are going to create is one that might be used if you want to install an NT service to a user account. When you are finished, the dialog will look like what is shown in Figure 12-26.

We will not go into the same amount of detail as we did when discussing the creation of the `BasicDialog` dialog. Here are the steps required to create the dialog resource.

1. If the `BasicDialog` is still in the project, export it to a dialog file. Then you can rename it to `InstallNTService`. Otherwise you can import the `BasicDialog` dialog and rename it. After the dialog is named, go to the Dialog Editor and add one check box control, two edit controls, and two text static controls that are used as labels for the two edit controls. Next, provide unique names for the new controls that you have added. The sizes used for the controls can be the same as those for similar controls

on other dialogs. To get the size of the controls on another dialog, right-click on the dialog and select Edit. Similar controls on different dialogs are sized the same to present a consistent look to the end user. You also want to make sure that none of the controls has an ID that is equal to the reserved numbers discussed in the last section.

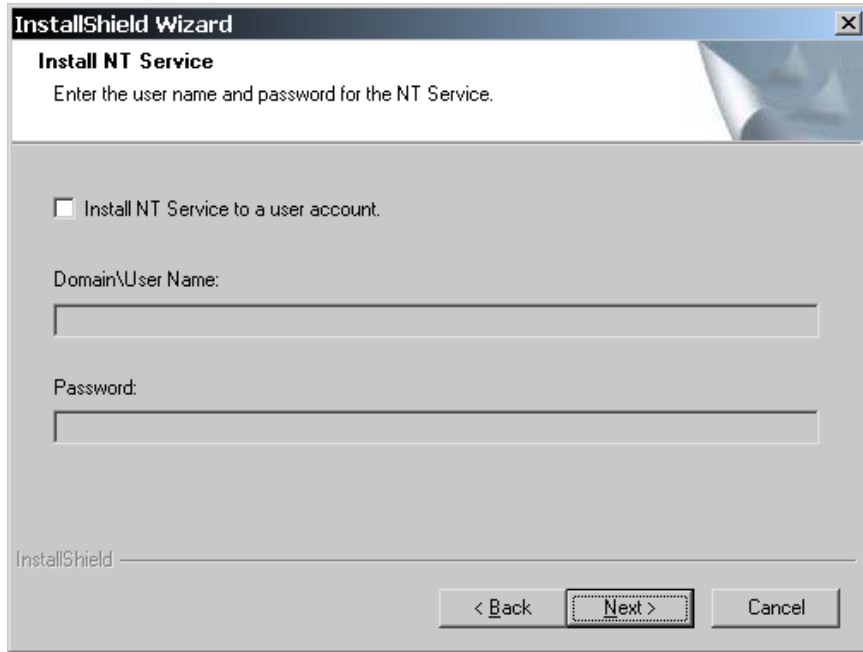


Figure 12-26: *The InstallNTService dialog.*

2. For the check box control, set the Sunken property to True. For the two edit controls, set the Enabled property to False so they are disabled until the user selects the check box. Set the Sunken property on the edit controls to True.
3. Set all the strings in the new and the old controls to unique string IDs. A possible format for the string IDs is to make the first part of the ID NTSERVICE_DIALOG_ and the last part of the ID the name of the control. You need to change even the string IDs that you inherited from the BasicDialog dialog to avoid string ID conflicts.

To create this new dialog, follow the steps listed above. These steps will create a dialog template, but the real work is in creating the dialog function. For this, you can use the `BasicDialog` function as the starting point.

1. Open the file `BasicDialog.rul` and resave it as `InstallNTService.rul`.
2. Change the name of the function to `InstallNTService`.
3. Set the value of the `szDlg` variable to the name of the dialog, which is the same as the name of the dialog function.
4. Follow the code example (Figure 12-27).

Before adding code to his function, you have to decide how this dialog is to work. For this example, the dialog's check box will be deselected, the edit controls disabled, and the Next button enabled. When the end user selects the check box, your code should enable the two edit fields and disable the Next button so the user cannot continue with NULL values for the `Domain\User Name` and `Password` edit fields. The Next button will be disabled until both edit fields contain values other than spaces. Also, if the user deselects the check box, the edit fields will be disabled again and any text that might be in them will be removed.

The strings that are entered into the two edit fields must be returned from the dialog function. This means that you need to change the function's prototype to have two additional string parameters that are typed as `BYREF STRING`. You need to add a few constants to the `MyCustDlgDefs.h` header file to identify the control IDs of the new controls in this dialog. Just as with string IDs, you should create unique constant names for the controls in a custom dialog. A format similar to the one used for string IDs would be good.

The source code for the dialog function that runs the `InstallNTService` dialog is shown in Figure 12-27. This code implements the functionality as described above.

```

////////////////////////////////////
//
// File Name:      InstallNTService.rul
//
// Description:    InstallScript file for a basic custom dialog
//                that can be used as a template.
//
// Comments:      This script demonstrates the creation
//                of an InstallScript custom dialog.
//
////////////////////////////////////

// prototypes of private functions
prototype INT RemoveSpaces(BYREF STRING);

////////////////////////////////////
// Function:      InstallNTService
//
// Purpose:       This function provides the functionality
//                behind the BasicDialog dialog.
//
////////////////////////////////////
function INT InstallNTService(szTitle, szSubTitle, svUser,
                              svPassword)
STRING  szDlg;
INT     nId;
HWND    hwndDlg, hwndTitle, hwndSubTitle;
HWND    hwndUserEdit, hwndPasswordEdit, hwndNext;
BOOL    bDone;
Begin

    // Define the name for the dialog to be used
    // in the custom dialog functions.
    szDlg = "InstallNTService";

    // Check the global variable to see if InstallScript
    // has already been initialized for custom dialogs.
    // Initialize the environment if not already initialized.
    if(!bSdInit) then
        SdInit();
    endif;

    // Define the dialog using the string name that is
    // used to identify it in the resource file.
    if(EzDefineDialog(szDlg, ISUSER, szDlg, 0) < 0) then
        MessageBox("Error defining dialog box", INFORMATION);
        abort;
    endif;

```

Figure 12-27: *The source code for the InstallNTService custom dialog box.*

```

// Initialize the message loop control variable.
bDone = FALSE;

// Loop in the dialog until the user takes action
// to close the dialog by clicking on a button.
while(!bDone)

    // Display the dialog and retrieve messages
    // based on the users interaction with the dialog.
    nId = WaitOnDialog(szDlg);

switch(nId)

    // The first message sent before the dialog
    // is displayed. This is where we can
    // initialize controls in the dialog.
    case DLG_INIT:

        // Get handles to the controls in the dialog.
        hwndDlg = CmdGetHwndDlg(szDlg);
        hwndUserEdit = GetDlgItem(hwndDlg,
                                INSTALLNTSERVICE_USEREDIT);
        hwndPasswordEdit = GetDlgItem(hwndDlg,
                                INSTALLNTSERVICE_PASSWORDEDIT);
        hwndNext = GetDlgItem(hwndDlg, CUST_BUTTON_NEXT);

        // NULL the strings for the two edit controls
        // to make sure that values are not persisted.
        SetWindowText(hwndUserEdit, "");
        SetWindowText(hwndPasswordEdit, "");

        // Set the custom title of the dialog.
        if(szTitle != "") then
            hwndTitle = GetDlgItem(hwndDlg, CUST_TEXT_TITLE);
            SetWindowText(hwndTitle, szTitle);
        endif;

        // Set the custom sub-title of the dialog.
        if(szSubTitle != "") then
            hwndSubTitle = GetDlgItem(hwndDlg,
                                CUST_TEXT_SUBTITLE);
            SetWindowText(hwndSubTitle, szSubTitle);
        endif;

```

Figure 12-27: *Continued.*


```

// Respond to the check box state being changed.
case INSTALLNTSERVICE_USERCHECKBOX:

    // If the check box is checked enable the edit
    // controls and disable the next button. If the
    // checkbox is unchecked then disable the
    // edit controls and enable the next button.
    if(CtrlGetState(szDlg, nId) = BUTTON_CHECKED) then
        EnableWindow(hwndUserEdit, TRUE);
        EnableWindow(hwndPasswordEdit, TRUE);
        EnableWindow(hwndNext, FALSE);
    elseif(CtrlGetState(szDlg, nId) =
            BUTTON_UNCHECKED) then
        EnableWindow(hwndUserEdit, FALSE);
        EnableWindow(hwndPasswordEdit, FALSE);
        EnableWindow(hwndNext, TRUE);
        SetWindowText(hwndUserEdit, "");
        SetWindowText(hwndPasswordEdit, "");
    else
        MessageBox("Unable to determine check box state",
                    INFORMATION);
    endif;

// Respond to changes in the user edit control
case INSTALLNTSERVICE_USEREDIT:

    // If the notification is that there has been a
    // change in the user edit control then get the text
    // and remove any leading and trailing spaces.
    if(CtrlGetSubCommand(szDlg) = EDITBOX_CHANGE) then
        CtrlGetText(szDlg, INSTALLNTSERVICE_USEREDIT,
                    svUser);

        RemoveSpaces(svUser);
        RemoveSpaces(svPassword);

        // If after removing leading and trailing spaces
        // and both the user and the password values are
        // not NULL then enable the next button.
        if(svUser != "" && svPassword != "") then
            EnableWindow(hwndNext, TRUE);
        endif;
    endif;

// Respond to changes in the password edit control
case INSTALLNTSERVICE_PASSWORDEDIT:

```

Figure 12-27: *Continued.*

```

// If the notification is that there has been a
// change in the password edit control then get the
// text and remove any leading and trailing spaces.
if(CtrlGetSubCommand(szDlg) = EDITBOX_CHANGE) then
    CtrlGetText(szDlg,
        INSTALLNTSERVICE_PASSWORDEDIT, svPassword);
    RemoveSpaces(svUser);
    RemoveSpaces(svPassword);

// If after removing leading and trailing spaces
// and both the user and the password values are
// not NULL then enable the next button.
if(svUser != "" && svPassword != "") then
    EnableWindow(hwndNext, TRUE);
endif;
endif;

// The code in this case statement responds to
// the end user clicking the Next button.
case CUST_BUTTON_NEXT:

    // Capture the values in the two edit controls.
    CtrlGetText(szDlg, INSTALLNTSERVICE_USEREDIT,
        svUser);
    CtrlGetText(szDlg, INSTALLNTSERVICE_PASSWORDEDIT,
        svPassword);

    RemoveSpaces(svUser);
    RemoveSpaces(svPassword);
    nId = CUST_BUTTON_NEXT;
    bDone = TRUE;

// The code in this case statement responds to
// the end user clicking the Cancel button or the
// Close button.
case CUST_BUTTON_CANCEL:

    Do(EXIT);

// The code in this case statement responds to
// the end user clicking the Back button.
case CUST_BUTTON_BACK:

    nId = CUST_BUTTON_BACK;
    bDone = TRUE;

```

Figure 12-27: *Continued.*

```

        // The code in this case statement responds to
        // an error that is found in the WaitOnDialog function.
        case DLG_ERR:
            SdError(-1, "InstallNTService");
            nId = -1;
            bDone = TRUE;

        default:

    endwhile;

endwhile;

// Close the dialog.
EndDialog(szDlg);

// Release memory used by the dialog.
ReleaseDialog(szDlg);

// Return the control ID of
// the control that was clicked.
return nId;

end;

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Function:      RemoveSpaces
//
// Purpose:       This function removes the leading and trailing spaces
//                in a string and returns the modified string.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
function INT RemoveSpaces(svStr)
STRING  svTemp;
INT     i, iLen;
BOOL    bDone;
begin
    // Get the length of the passed string.
    iLen = StrLengthChars(svStr);
    // If the string is NULL return.
    if (iLen = 0) then
        return ISERR_SUCCESS;
    endif;

    // Initialize the index.
    i = 0;

```

Figure 12-27: *Continued.*

```

// Remove the leading spaces.
while ((i < iLen) && (!bDone))
    if ((svStr[i] = '\t') || (svStr[i] = 32)) then
        i = i + 1;
    else
        bDone = TRUE;
    endif;
endwhile;

// Get the length of the modified string without
// the leading spaces.
iLen = StrSub(svStr, svStr, i, iLen-i);

// Remove the trailing spaces
while ((iLen > 0) && (!bDone))
    if ((svStr[iLen-1] = '\t') || (svStr[iLen-1] = 32)) then
        iLen = iLen - 1;
    else
        bDone = TRUE;
    endif;
endwhile;

// Add a NULL terminator.
svStr[iLen] = 0;

return ISERR_SUCCESS;
end;

```

Figure 12-27: *Continued.*

In the `InstallNTService` dialog function, some code is added to the `DLG_INIT` case statement to get the handles to the two edit controls and to the Next button control. The function uses these handles to enable and disable these controls. The function also contains two statements that null out the string values for the two edit fields so these values do not persist when we reenter the dialog box after jumping back to the previous dialog and to prevent values being passed to the function. Case statements are added for the check box control and the two edit controls.

Under the case statement for the check box control, the `CtrlGetState` `InstallScript` function determines if the check box is selected or deselected. Depending on whether the end user has selected the check box, the function enables or disables the user name and password edit controls and the Next button. The Windows API `EnableWindow` enables and disables these controls. The `ifx.h` header file includes the header file that prototypes this Windows API. For the

scenario where the user starts to enter values in the two edit controls and then deselects the check box, the function nulls the values entered into the user name and password edit controls. Under the case statement for the Next button, the function captures the values entered into the user name and password edit controls using the InstallScript `CtrlGetText` function. The values that are entered for user name and password are passed back to the calling function, which in most cases will be the `OnFirstUIBefore` event handler. Note that a private function called `RemoveSpaces` was added to the `InstallNTService.rul` file. This function removes leading and trailing spaces so the user cannot just type in spaces and continue with the installation.

You can test this new dialog in the same fashion you used to test the `BasicDialog` dialog. Make sure that you export this dialog because you will use it when we discuss the user interface in a Basic MSI project.

We have now covered the basics of creating custom dialogs in a Standard project. Before we move on to a discussion of creating the user interface in a Basic MSI project, we need to discuss one extension to the basics of custom dialogs in Standard projects. This extension is how to handle a custom dialog when using a response file to perform a silent installation.

Handling Silent Installations

One of the approaches to performing a silent installation is to have the script read a file, called a response file, that contains the default values to be used as input to the dialogs in the user interface. The trick is to prevent the dialogs from being displayed. To do this, each of the dialog functions needs to recognize that a silent install is running and read the default input from the response file and then return from the dialog function before the lines of code that create and display the dialogs are executed.

To generate a response file, you have to add code to the dialog functions. The code to generate the response file is added at the end of the dialog function. The code to implement silent installs for the `InstallNTService` dialog is shown in Figure 12-28. The code shown in this figure is what is placed at the beginning and at the end of the `InstallNTService` dialog function.

```

////////////////////////////////////
// Function:      InstallNTService
//
// Purpose:       This function provides the functionality
//                behind the BasicDialog dialog.
////////////////////////////////////
function INT InstallNTService(szTitle, szSubTitle, svUser,
                              svPassword)

STRING  szDlg, svDataKey, svVal;
INT     nId, nvInstallNTService, nvVal;
HWND    hwndDlg, hwndTitle, hwndSubTitle;
HWND    hwndUserEdit, hwndPasswordEdit, hwndNext;
BOOL    bDone;
begin

    // Define the name for the dialog to be used
    // in the custom dialog functions.
    szDlg = "InstallNTService";

    // Read the data produced by this dialog when
    // the response file was created.
    if(MODE=SILENTMODE) then
        SdMakeName(svDataKey, szDlg, "", nvInstallNTService);
        SilentReadData(svDataKey, "Result", DATA_NUMBER, svVal, nId);

        if((nId != BACK) && (nId != CANCEL)) then
            SilentReadData(svDataKey, "szUser", DATA_STRING,
                          svUser, nvVal);
            SilentReadData(svDataKey, "szPassword", DATA_STRING,
                          svPassword, nvVal);
        endif;

        // Return before displaying the dialog.
        return nId;
    endif;

    .
    .
    .
    // Insert dialog function code from Figure 12-27 here.
    .
    .
    .

    // Close the dialog.
    EndDialog(szDlg);

```

Figure 12-28: *The code for implementing silent installs using a response file.*

```

// Release memory used by the dialog.
ReleaseDialog(szDlg);

// Record the data produced by this dialog in a response file.
if(MODE=RECORDMODE) then
    SdMakeName(svDataKey, szDlg, "", nvInstallNTService);
    SilentWriteData(svDataKey, "Result", DATA_NUMBER, "", nId);
    SilentWriteData(svDataKey, "szUser", DATA_STRING, svUser, 0);
    SilentWriteData(svDataKey, "szPassword", DATA_STRING,
                    svPassword, 0);

endif;

// Return the control ID of
// the control that was clicked.
return nId;
end;

```

Figure 12-28: *Continued.*

There are three InstallScript functions that implement support for silent installations: `SdMakeName`, `SilentWriteData`, and `SilentReadData`. The `SdMakeName` function is used to generate part of the section name that is written to the response file for each dialog in the user interface sequence. The `SilentWriteData` function is used to write the values into the response file that are to be part of the default install when it is run in silent mode. This function also takes the name created by the `SdMakeName` function and completes the section name by using the value of the `ProductCode` for the application being installed as the first part of the section name in the response file. The `SilentReadData` function is used at the beginning of the dialog function to read the response file. After the data for the dialog is read, the code exits the dialog function before the `EzDefineDialog` or the `WaitOnDialog` functions are called.

If you add the code shown in Figure 12-28 to the `InstallNTService` dialog function and run the installation in record mode, it generates a response file that is created in the `%WINDOWS%` folder. If you accept all of the defaults in the user interface for the Developer Art application, a response file shown in Figure 12-29 is generated.

```
[{D375D664-6EF0-4495-9762-68AB33CC3D83}-DlgOrder]
Dlg0={D375D664-6EF0-4495-9762-68AB33CC3D83}-SdWelcome-0
Count=6
Dlg1={D375D664-6EF0-4495-9762-68AB33CC3D83}-SdCustomerInfo-0
Dlg2={D375D664-6EF0-4495-9762-68AB33CC3D83}-InstallNTService-0
Dlg3={D375D664-6EF0-4495-9762-68AB33CC3D83}-SetupType-0
Dlg4={D375D664-6EF0-4495-9762-68AB33CC3D83}-SdStartCopy-0
Dlg5={D375D664-6EF0-4495-9762-68AB33CC3D83}-SdFinish-0
[{D375D664-6EF0-4495-9762-68AB33CC3D83}-SdWelcome-0]
Result=1
[{D375D664-6EF0-4495-9762-68AB33CC3D83}-SdCustomerInfo-0]
szName=Bob Baker
szCompany=InstallShield
Result=1
[{D375D664-6EF0-4495-9762-68AB33CC3D83}-SprintfBox-0]
Result=6
[{D375D664-6EF0-4495-9762-68AB33CC3D83}-SdFinish-0]
Result=1
bOpt1=0
bOpt2=0
[{D375D664-6EF0-4495-9762-68AB33CC3D83}-InstallNTService-0]
Result=1
szUser=' '
szPassword=' '
[{D375D664-6EF0-4495-9762-68AB33CC3D83}-SetupType-0]
Result=301
[{D375D664-6EF0-4495-9762-68AB33CC3D83}-SdStartCopy-0]
Result=1
```

Figure 12-29: *The response file for the Developer Art application.*

This completes the discussion about creating custom dialogs in a Standard project. This did not cover all the issues that might be faced, but it provides a good basis in how to approach the subject. Further help can be obtained by looking at the source code for all the dialogs that are part of a new Standard project. The .rul files for these dialogs can be found in the following locations:

```
C:\Program Files\InstallShield\Developer\Script\isrt\src
```

and

```
C:\Program Files\InstallShield\Developer\Script\iswi\src
```


Generating a User Interface for a Basic MSI Project

In this section, you will use the `DeveloperArt_IDEMSI.ism` project file to explore how the user interface is created in a Basic MSI project. The operation of the Dialog Editor in a Basic MSI project is the same as in the Standard project. The only difference is that when you place a control that requires end user interaction, other than push buttons, there is a public property name that has to be entered as part of the control attributes. You are going to import the `InstallNTService.isd` file, add the property names for the appropriate controls, and insert the dialog into the user interface that is displayed during a fresh installation.

As discussed, dialog creation in a Basic MSI project does not require a resource DLL or the development of a dialog function. Everything that you create in the Dialog Editor is entered into database tables and, during the installation, the Windows Installer is responsible for reading the database tables and generating the correct dialogs. The main focus here is to discuss how to develop the same functionality in a Basic MSI project that you created using a dialog function in a Standard project.

Implementing the InstallNTService Dialog

Load the `DeveloperArt_IDEMSI` project and go to the Dialogs view and import the `InstallNTService` dialog. During the import process, the Resolve Conflict dialog appears (Figure 12-30). The source of this conflict is in the Binary table where, in a Basic MSI project, all the bitmaps and icons displayed in the dialogs are contained as entries. When you created the `InstallNTService` dialog in the Standard project, the banner bitmap used at the top of the dialog was streamed into the Binary table using the identifier `NewBinary1`. When you create a Basic MSI project, there is already a row in the Binary table that uses the `NewBinary1` identifier so you need to rename the identifier for the imported dialog.

To rename the identifier, select the Rename radio button in the Resolve Conflict dialog and then click Edit. This displays a dialog where you can enter a unique name for the binary stream that is being imported.

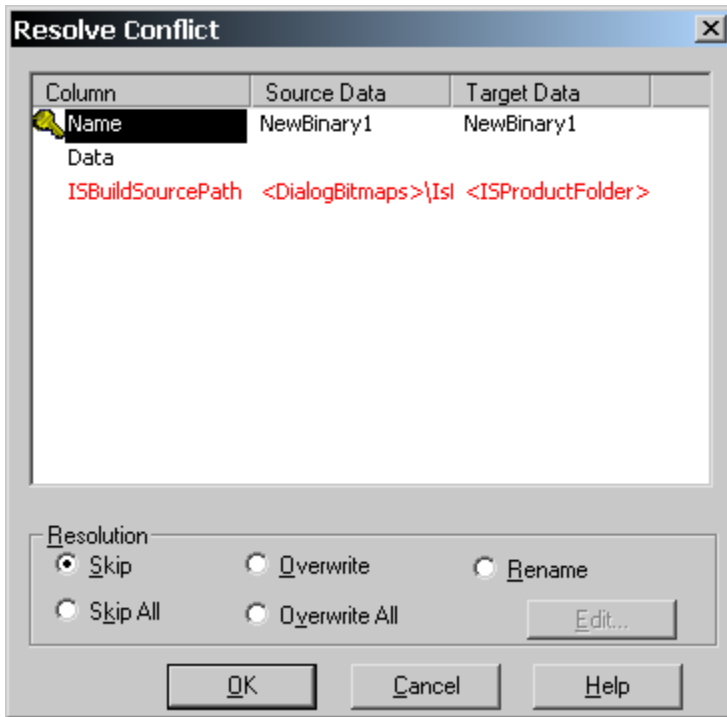


Figure 12-30: Conflict resolution when importing a dialog file.

When the import process is complete, you need to make changes to some of the attributes for some of the controls. The changes you need to make are discussed in the following list.

Dialog Caption: In a Standard project, the caption of the dialogs displays the string "InstallShield Wizard". In a Basic MSI project, the caption includes the name of the product in addition to the "InstallShield Wizard" string. To make this dialog look the same, you have to modify the caption so it includes the name of the product being installed. The modification is made using the String Table dialog and the new string to be entered is "[ProductName] - InstallShield Wizard". When the installation is run, the value of the ProductName property is substituted for the property name inside the square brackets.

BannerBitmap: For the bitmap control, set the Stretch to Fit property's value to True. If you do not do this, the bitmap will not be sized to the same size as the dialog at screen resolutions different than the build machine.

Title Text Control: As you saw in the Standard project, the Title text static control had a control identifier of 50, so when it was created in the dialog function it was displayed in bold text. In a Basic MSI project, you need to specifically set the text style to be used when the text is displayed. Do this by selecting `TahomaBold8` in the Text Style property's value column. The title text will appear in bold.

UserCheckBox Control: For the check box control, you need to define the name of a public property that is set to a value when the control is selected by the end user. You also have to define the value to which the property is set. In the Property property, enter the name of a property in all uppercase letters. In this example, you can use `USERACCOUNT` as the property. To define the value to which it gets set, go to the Value property and enter the value. For this example, use `1` as the value. This property can be used in conditions. Finally you need to set the Sunken property to `False`. Because of the way Windows Installer implements the check box control when the Sunken property is set to `True`, both the check box and the label are sunken. This makes for an unattractive control.

UserEdit Control: To retrieve the text that a user enters into an edit control, you need to assign to the control the name of a public property. For this control, use `ACCOUNT` as the property. To assign a property to an edit control, go to the Property property and enter the name of the property.

PasswordEdit Control: For this edit control, you also need to enter the name of a property in the Property value field. For this edit control, use `PASSWORD` as the property.

BrandingLine Control: In a Basic MSI project, the dialog box branding is added during the build process. Accordingly you need to remove this control from the imported `InstallNTService` dialog. To remove a control, select the control and press the Delete keyboard button.

Next, Back, and Cancel Push Button Controls: In a Basic MSI project, the size and location of these push buttons are different than in a Standard project. So that this dialog looks the same, you need to make the following changes in the size and location of the buttons.

- **Width property:** The width of all buttons is `66`.

- **Top** property: The top of all buttons should be 243.
- **Left** property: Back – 164, Next – 230, Cancel – 301

Final Modification: After you have made the changes described above, go to the Property Manager under Advanced Views and remove the USERACCOUNT, ACCOUNT, and PASSWORD properties. Otherwise, they will be built into the Property table and the dialog will have the check box checked and values in the two edit controls when the dialog is displayed.

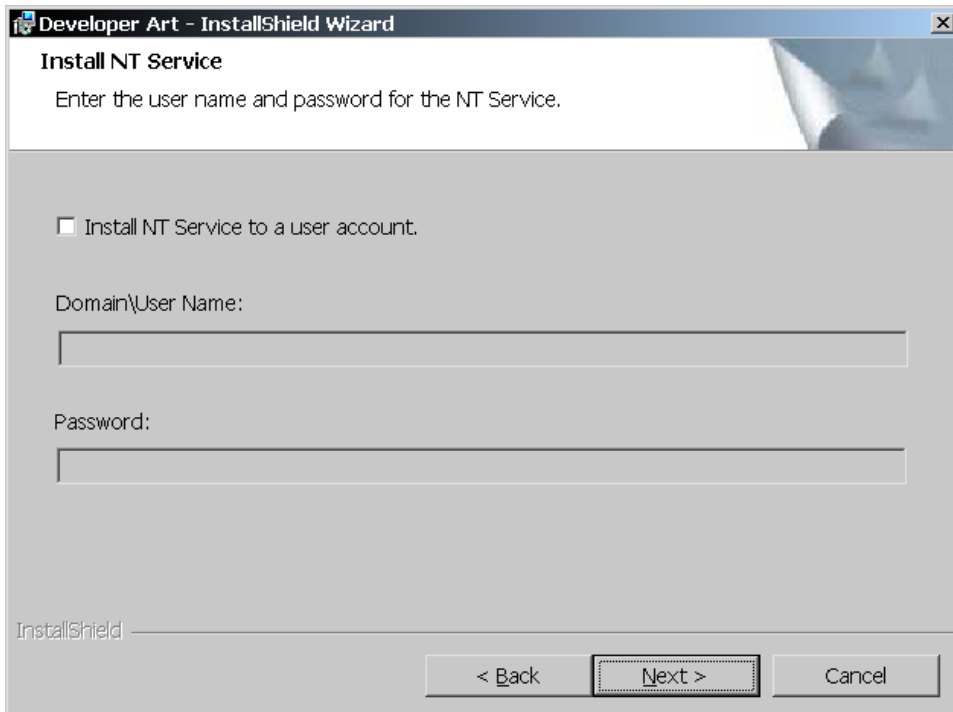


Figure 12-31: *The InstallNTService dialog as implemented in a Basic MSI project.*

When you make the above changes, you will create a dialog that looks like what is shown in Figure 12-31.

Now that you have created a dialog for use in a Basic MSI project, you need to do something with it. The first thing that you need to do is insert it into the user interface sequence so that when an end user performs a fresh installation, the dialog appears

after the CustomerInformation dialog and before the SetupType dialog. This involves the use of control events. Control events can be considered messages that a dialog sends to the Windows Installer, specifying that a certain action take place.

We begin by looking at the default sequence of dialogs that are displayed when an end user runs a fresh installation. You can examine this sequence by going to the Sequences view under Step 5 and expanding the User Interface tree under the Installation folder. Move down the list of actions and dialogs until you get to the InstallWelcome dialog entry in the InstallUISequence table. Expand the tree under this dialog to see what is shown in Figure 12-32.

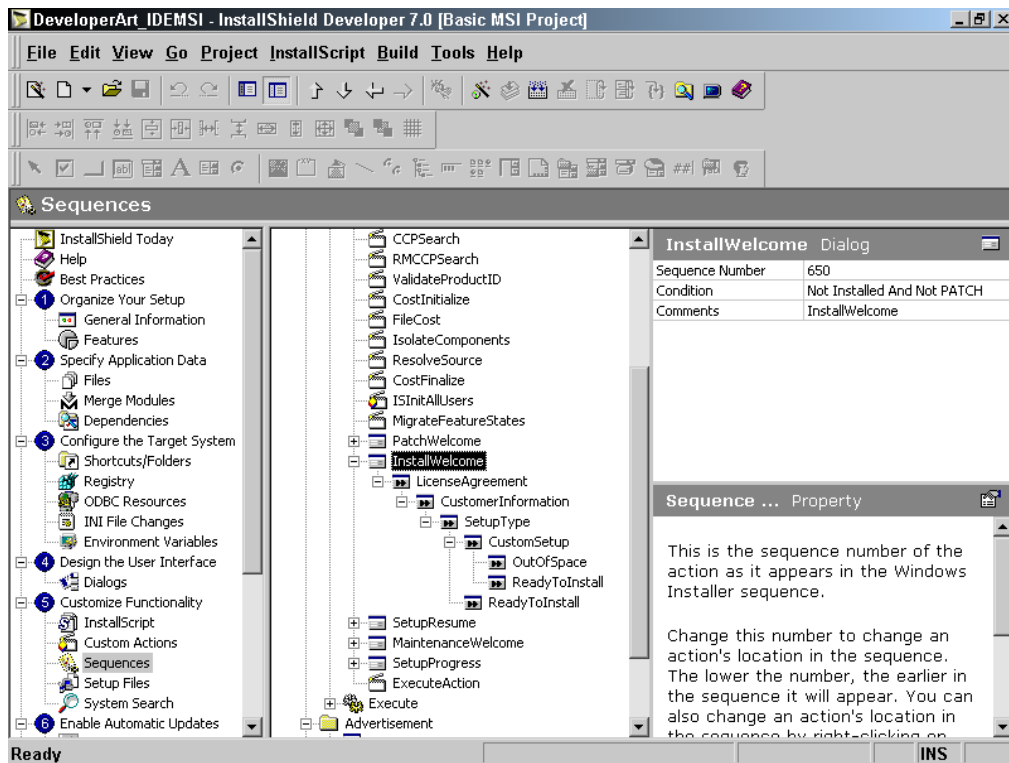


Figure 12-32: The default sequence of dialogs for the fresh install of a Basic MSI project.

To review how this works, the Windows Installer executes in order of sequence number all actions and dialogs that have a condition that evaluates to TRUE. During a fresh installation, the Windows Installer reaches the InstallWelcome dialog, finds that the condition evaluates to TRUE, and displays the dialog. Since the

InstallWelcome dialog is a modal dialog, nothing more can happen in the installation until the end user dismisses the dialog. The user can click the Cancel button, which terminates the installation, or they can click the Next button, which displays the LicenseAgreement dialog.

The Next button launches the LicenseAgreement dialog because there is a control event called NewDialog assigned to the button. The argument to this control event is the name of the new dialog that is to be displayed when the control event is fired. In order to implement a wizard sequence in which the end user can move back and forth, the Back button on the LicenseAgreement dialog fires the NewDialog control event that has the InstallWelcome dialog as its argument. This is the same scenario for all the dialogs in the wizard sequence until the final dialog is reached. The button on the ReadyToInstall dialog fires a control event named EndDialog with the Return argument. This control event destroys the dialog that is currently being displayed and returns control back to the Windows Installer so the process can continue executing the actions and dialogs in the sequence table.

To insert the InstallNTService dialog between the CustomerInformation and SetupType dialogs, you need to redirect the NewDialog control events attached to the Next and Back buttons on the affected dialogs. Starting with the InstallNTService dialog, go to the Behavior icon under the dialog in the Dialogs view. This displays a list of the controls that make up the dialog (Figure 12-33).

1. Select the Next button from the list of controls and click in the Event column.
2. Select the NewDialog control event from the drop-down menu.
3. In the Argument column, select from the list of dialogs the SetupType dialog.
4. In the Condition column, enter the number 1, which serves to make the condition always TRUE.
5. In similar fashion for the Back button, assign a NewDialog control event that points at the CustomerInformation dialog.
6. For the Cancel button, you need to select a different control event. You do not want to dismiss the current dialog, but display a confirmation

dialog that is a child of the current dialog. To do this, select the SpawnDialog control event and make its argument the CancelSetup dialog.

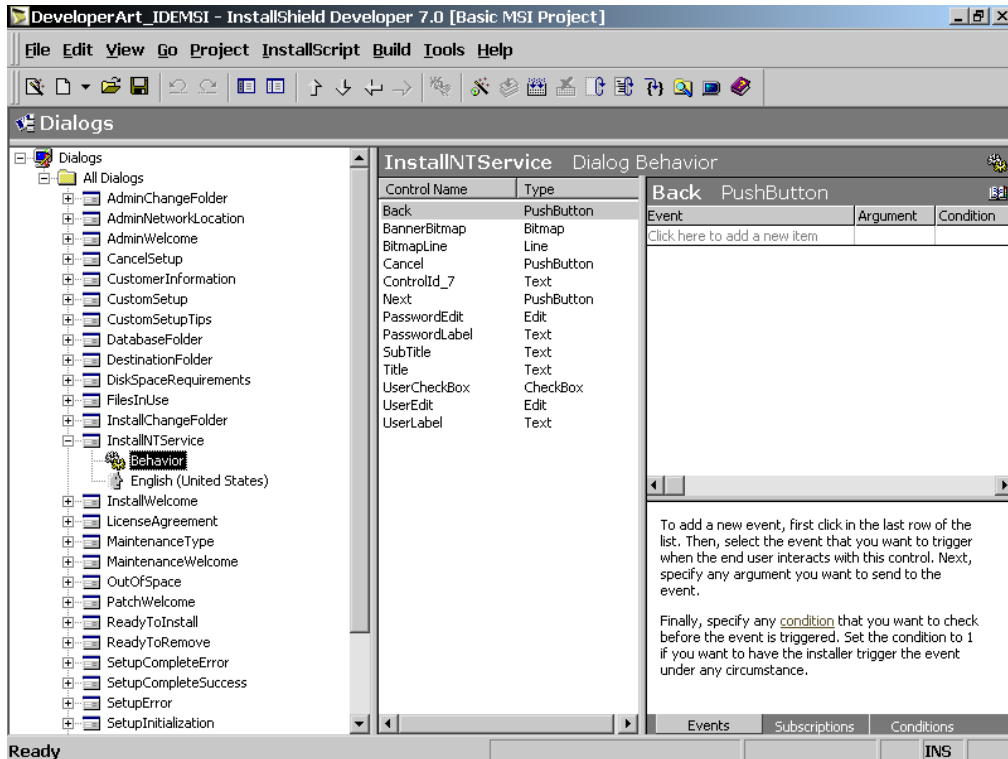


Figure 12-33: *Specifying the behavior of controls in the Dialogs view.*

You now have the InstallNTService dialog configured properly, but there is no dialog yet that will display it.

To insert this dialog into the wizard sequence:

1. Go to the CustomerInformation dialog and change the argument for the NewDialog control event attached to the Next button to be the InstallNTService dialog.
2. For the SetupType dialog, change the argument for the NewDialog event for the Back button to be the InstallNTService dialog. Once this

operation is complete, you have inserted the InstallNTService dialog into the wizard sequence that is displayed during a fresh installation.

You can build the project and verify that everything works as expected by running the installation using the Test button in the toolbar.

What you have accomplished so far is to just insert the dialog into the sequence of dialogs. This does not implement the same functionality that was created in the Standard project using the dialog function. The first thing that you need to do is to enable and disable the UserEdit and PasswordEdit edit controls when the end user selects or deselects the check box control. To implement this functionality:

1. Go to the Conditions tab under the Behavior icon for the InstallNTService dialog box. Select the UserEdit control and enter two control conditions as shown in Figure 12-34.

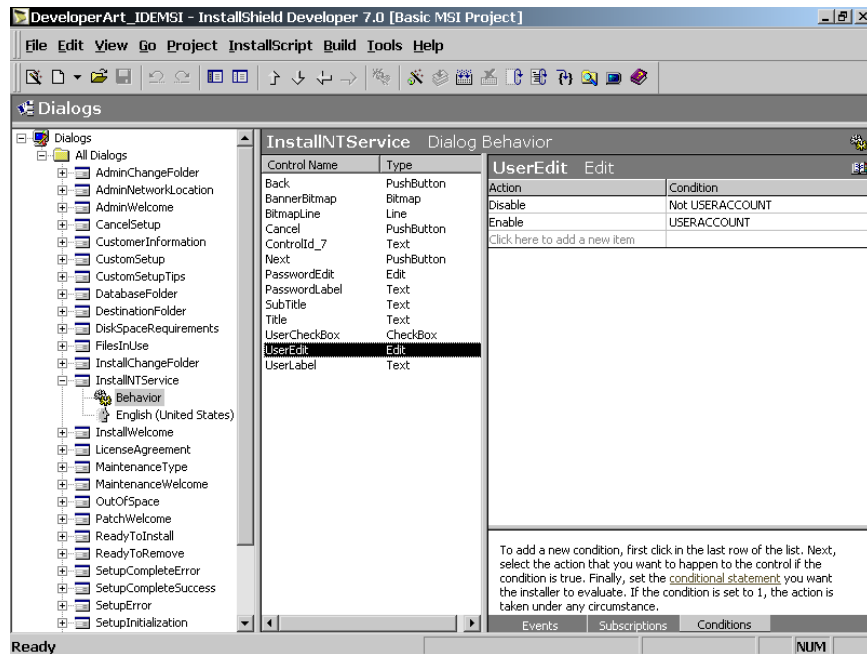


Figure 12-34: Setting control conditions for the UserEdit control.

2. As the condition, use the USERACCOUNT property, which is tied to the UserCheckBox control. Disable the UserEdit control when the

USERACCOUNT property does not have a value and we enable the UserEdit control when the USERACCOUNT property does have a value. The USERACCOUNT property has a value of 1 when the UserCheckBox control is selected and it has no value when the UserCheckBox control is not selected.

3. Select the PasswordEdit control and make the same two entries under the Conditions tab as shown in Figure 12-34.

You can build the project to verify that when the end user selects the check box in the dialog, the edit controls are enabled. When the end user deselects the check box, the edit controls are disabled again.

Now you will implement some functionality to prevent the end user from continuing with installation when the check box is selected, but both edit controls do not have entries. It might seem that you could just do what you did for the edit controls and place some control conditions on the Next button. You could disable the Next button when the user selects the check box control and enable it again when the user has entered values in both of the edit controls. The problem, however, is that the Windows Installer implementation of the edit control is such that changes made in the control are not recognized. Because of this, even though the end user typed values into both the UserEdit control and the PasswordEdit control, there would be no message sent to force the reevaluation of the control conditions that you placed on the Next button.

You could ask the end user to click in the first edit control in which values were entered, or deselect and select again the check box control, or go back to the previous dialog and come forward again. Any one of these actions would force the reevaluation of the control conditions and enable the Next button. However, this is not very intuitive for the user and definitely non-Windows. The best approach to take is to place a condition on the NewDialog control event assigned to the Next button and to display a message box telling the user that both edit controls need to have values before they can continue with the installation.

The condition that you place on the NewDialog control event for the Next button needs to verify that if the check box is selected, both edit fields have values entered. If the check box is not selected it does not matter whether values were entered into the edit controls. The following condition does this.

Not USERACCOUNT Or (USERACCOUNT And ACCOUNT And PASSWORD)

If you leave it at this, the end user will not know what the problem is when they click the Next button and nothing happens because they have not filled in both edit controls. To display a message box to the end user instructing them to fill in both edit controls, you can use one of two methods. You could create a new dialog in the Dialog Editor with a message in it. This dialog would be launched by a SpawnDialog control event attached to the Next button. With less work, you can create a simple custom action that you can attach to the Next button using the DoAction control event. Based on the appropriate condition, it will be executed when the end user tries to continue the installation without filling both edit fields. This is the approach that you will use in this example.

First, create an InstallScript custom action that will display a message. So this message can be localized if required, identify this message with a string ID in the string table. The code for this custom action is shown in Figure 12-35.

```

////////////////////////////////////
//
//  I I I I I I I I  S S S S S S
//  II   SS                               InstallShield (R)
//  II   S S S S S S                               (c) 1996-2001,
//  II   SS                               InstallShield Software Corporation
//  I I I I I I I I  S S S S S S                               All rights reserved.
//
//
//
//      File Name:      Setup.rul
//
//      Description:    InstallShield script for defining custom
//                      actions in a Basic MSI project.
//
////////////////////////////////////

// Include files for custom actions.
#include "isrt.h"
#include "iswi.h"

// Prototypes for functions that are targets of custom actions.
export prototype InstallNTServiceMsg(HWND);

```

Figure 12-35: *Setup.rul* for the message custom action used in the *InstallNTService* dialog.

```

////////////////////////////////////
//
// Function:      InstallNTServiceMsg
//
// Purpose:      This function displays a message informing
//               the end user that both edit controls need
//               to be filled in if an NT service is to be
//               installed to a user account.
//
////////////////////////////////////
function InstallNTServiceMsg(hMSI)
begin
    MessageBox(@NTSERVICE_DIALOG_WARNINGMSG, INFORMATION);
end;

```

Figure 12-35: *Continued.*

For this custom action to work, you need to go to the String Table Editor under the General Information view and enter an appropriate message with a string ID `NTSERVICE_DIALOG_WARNINGMSG`. Using the `@` symbol in front of the string ID in `InstallScript` substitutes the value of the referenced string ID. After creating the code for the custom action, create the custom action as discussed in Chapter 10. The name used for the custom action in this example is `InstallNTServiceMsg`.

Once the custom action is created, you need to attach it to the Next button using the `DoAction` control event. The argument to this control event is the name of the custom action that you want executed. Next, you need to place a condition on the `DoAction` control event that is the complement of the condition that placed on the `NewDialog` control event. The following condition will do this.

```

USERACCOUNT And ((ACCOUNT Xor PASSWORD) Or (Not ACCOUNT And
                                                         Not PASSWORD))

```

This condition causes the execution of the custom action if the `UserCheckBox` control is selected and neither of the edit controls has a value entered, or only one of the edit controls has a value entered.

There is one last scenario that you need to handle. This is where the user starts to enter values into one or both of the edit controls and then decides to deselect the `UserCheckBox` control. In this case, if you do nothing, one or both of the `ACCOUNT` and `PASSWORD` properties will have values. What you want is to allow

these properties to have values only when the NT service is to be installed to a particular user's account. Otherwise, these properties need to be NULL. To implement this functionality, add two more control events to the Next button. These two control events are the SetProperty control event. This name does not appear in the list of control events in the Dialog Editor because this control event is implemented in a special fashion. To use this control event to set the value of a property, place the name of the property inside square brackets in the Event column and place the value to which you want to set the property into the Argument column.

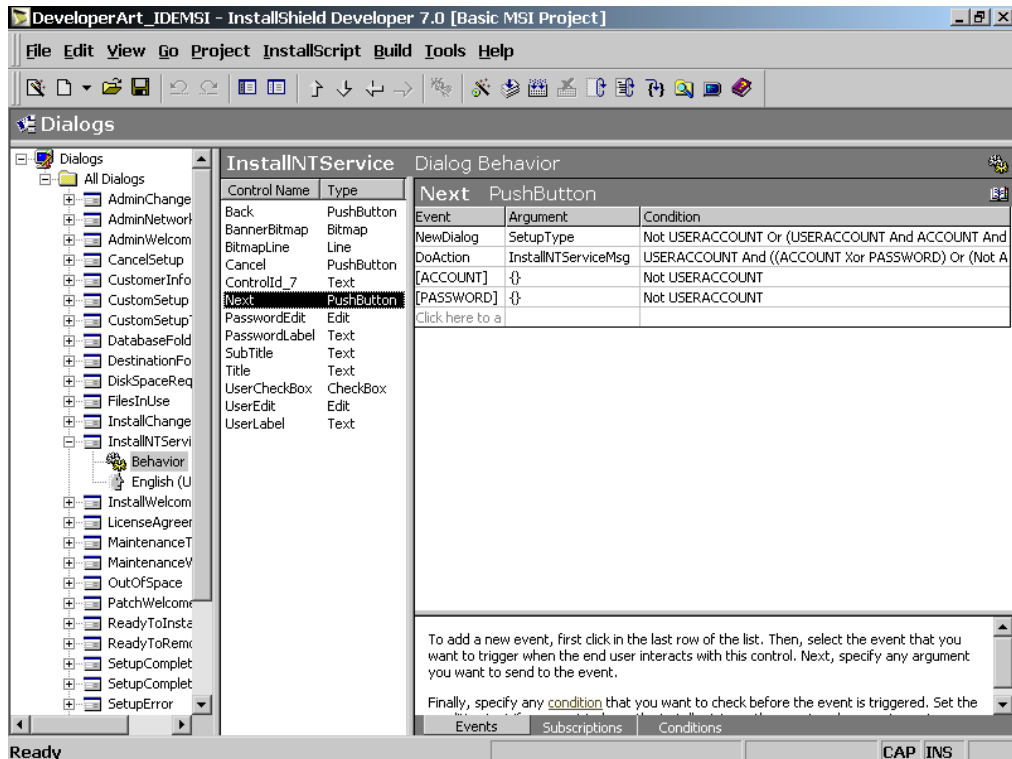


Figure 12-36: *The final set of control events for the Next button in the InstallNTService dialog.*

To set the value of a property to NULL, place a pair of curly braces ({}) into the Argument column. Therefore, to set the value of the ACCOUNT property and the PASSWORD property to NULL, you need to place these two properties inside square brackets in the Event column for the Next button and place a pair of curly braces in the Argument column. You want to set the value of these two properties to

NULL only if the UserCheckBox control is not selected. The condition for this is as follows:

```
Not USERACCOUNT
```

The control events for the Next button in the InstallNTService dialog box should now look like what is shown in Figure 12-36.

When the Windows Installer performs the control events for a control, it executes all of the events as long as the condition evaluates to TRUE. However, it executes the control events in the order they are entered into the Event column. To reposition control events, right-click on the event that you want to move and select Move Up or Move Down.

You have now implemented in a Basic MSI project the InstallNTService dialog with the same functionality as created in the Standard project. Since there are a number of changes you made to the dialog that was created in the Standard project, you can export this dialog to a dialog file. In order to preserve the dialog that you created in the Standard project, name this exported file InstallNTService_MSI.isd.

Adding Serial Number Input to the CustomerInformation Dialog

In a Standard project there are two CustomerInformation dialogs, one without a control for entering serial numbers and one with such a control. In a Basic MSI project there is only one CustomerInformation dialog and it has a control for the user name, the company name, and serial number input. By default, the control for the serial number is not visible. In this section, you will enable the input of a serial number in the CustomerInformation dialog and then implement the Windows Installer mechanism for dealing with serial numbers.

First, we need to discuss the built-in functionality in Windows Installer for working with serial numbers. This mechanism is not very robust and is useful only for stopping the installation when the end user is not knowledgeable about how the Windows Installer works. The basic philosophy of Microsoft and the designers of the Windows Installer technology is that security needs to reside with the application itself and not in the installation. The MaskedEdit control in Windows Installer is used

for serial number input. A MaskedEdit control is an edit control that contains a mask that defines the separate edit windows into which the user types the serial number. In addition to the MaskedEdit control, the Windows Installer functionality for serial numbers is implemented by three properties, one control event, and one standard action. The properties are PIDTemplate, PIDKEY, and ProductID. ValidateProductID is the name of both the control event and the standard action.

First, this section looks at the MaskedEdit control to see how several of the properties come into play. The mask is used to define how a MaskedEdit control is displayed at run time (Figure 12-37).

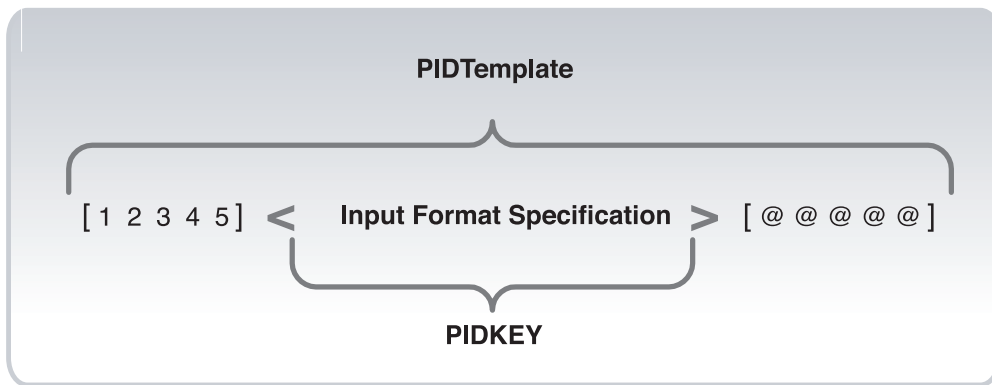


Figure 12-37: *Specifying the mask for a MaskedEdit control.*

The mask for a MaskedEdit control is defined by the PIDTemplate property. The mask that this property defines consists of three distinct sections, two of which are optional. With reference to Figure 12-37, the required section of a mask is that which is bounded by the angle brackets. Inside the angle brackets, you specify various input format characters that determine the type of input that is allowed. What is inside the angle brackets determines the visible part of the mask, and whatever the user enters here becomes the value of the PIDKEY property. Outside the angle brackets you can optionally specify numbers or letters that are treated as literal constants or specify the @ symbol which the Windows Installer will replace with a randomized digit.

When the user enters values into the MaskedEdit control, the value of the PIDKEY property is set. After the serial number input, the installation then executes either the ValidateProductID control event or standard action to compare the value of the PIDKEY property against the input format specification. If the input is validated, the

ValidateProductID control event or standard action sets a value for the ProductID property that is equal to the value entered by the end user plus the characters or random numbers defined outside the angle brackets.

There are a number of characters that you can use to create the input format specification. These characters and their meanings are described in Table 12-4.

Table 12-4: MaskedEdit Control Format Specification Characters

Character	Meaning
#	Indicates that only a digit can be entered at its location.
%	Alternate character for specifying that only a digit can be input at that location.
_	Indicates that an alphanumeric character can be entered at its location.
^	Alternate character for specifying that an alphanumeric character can be entered at its location.
?	Alternate character for specifying that an alphanumeric character can be entered at its location.
'	Alternate character for specifying that an alphanumeric character can be entered at its location.
-	Used to define the separation between one edit window and another. For any particular edit window, you can use only one of the formatting characters described in the previous rows of this table. This character must separate different characters. This creates a separate edit window.

Now that you know how the Windows Installer works when it comes to serial number, you can implement this capability in the CustomerInformation dialog. The first thing that you have to do is make some changes in the dialog so the serial

number MaskedEdit control becomes visible. When you do this, you will also be removing certain entries that are applicable only to a project that has been upgraded from InstallShield Express.

The changes that you need to make affect only two controls. The changes you need to make are described in the following steps:

1. For the SerialLabel Text control, set the Visible property to True. In addition, go to the Behavior Editor and delete the Show control condition.
2. For the SerialNumber MaskedEdit control set the Visible property to True, set the Mask property to [PIDTemplate], and modify the value of the Property property to be PIDKEY. In addition, go to the Behavior Editor and delete the Show control condition and go to the Property Manager under Advanced Views and delete the PIDKEY property so that it does not have an initial value.
3. Define the PIDTemplate property so it conforms to the serial number format that you want to use. You can use the same format used in Chapter 8 where there are three fields. The first field contains six letters that define the product, the second contains four digits that define the version of the product, and the third contains ten digits that represent a sequence number that makes each serial number unique. A possible value for the PIDTemplate property would then be as follows:

```
12345<??????-####-#####>@@@@@
```

In the Property Manager, there is already a default value for the PIDTemplate property so all you have to do is change the default value to what is shown above. When you make this change, build the project, and run the installation, you should see a CustomerInformation dialog that looks like what is shown in Figure 12-38.

In Figure 12-38 the serial number input control does not have a three-dimensional look. This is because, for a MaskedEdit control, setting the Sunken property to True generates an unattractive control so it is customary to leave the Sunken property set as False.

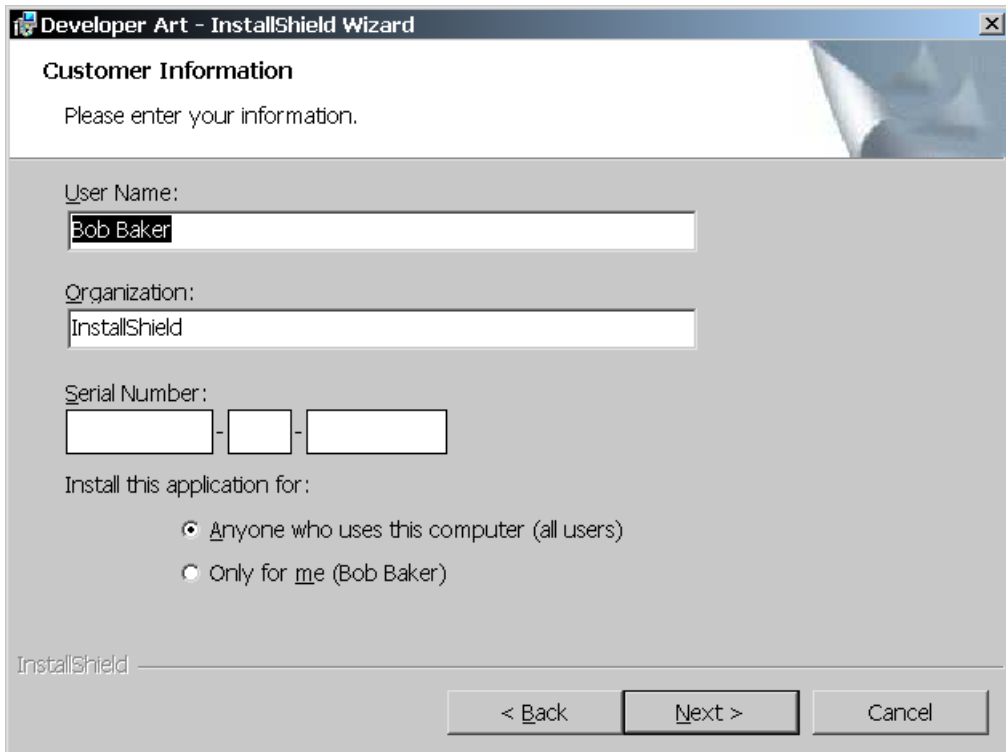


Figure 12-38: *The modified CustomerInformation dialog to include serial number input.*

Now that you have an operating control that can receive serial number input, you need to do something so the user cannot continue past the CustomerInformation dialog unless the serial number is entered. This functionality is implemented through the appropriate use of control events attached to the Next button. You need to go to the Dialog Editor and then to the Behavior icon and make the following changes to the control event for the Next button.

1. Delete the EndDialog control event because it is not required. This is a valid control event only for a project that is upgraded from InstallShield Express.
2. Change the condition for the NewDialog control event to be the ProductID property.

3. Add ValidateProductID as a new control event with 0 as the argument and 1 as the condition. This particular control event does not require an argument, but the Argument column of the ControlEvent table cannot be NULL, so enter 0 for this value. After adding this control event, you need to move it so it is executed before the NewDialog control event. This is because you need to create a value for the ProductID property before running the NewDialog control event since it is being used as the condition.
4. In the Property Manager under Advanced Views, delete the ProductID property. This prevents this property from being persisted in the database. If this property is persisted in the database, it must have a value equal to the string "none". If it is persisted, then when you set it using the ValidateProductID control event, the value that is eventually written to the registry will still be the persisted value of "none".

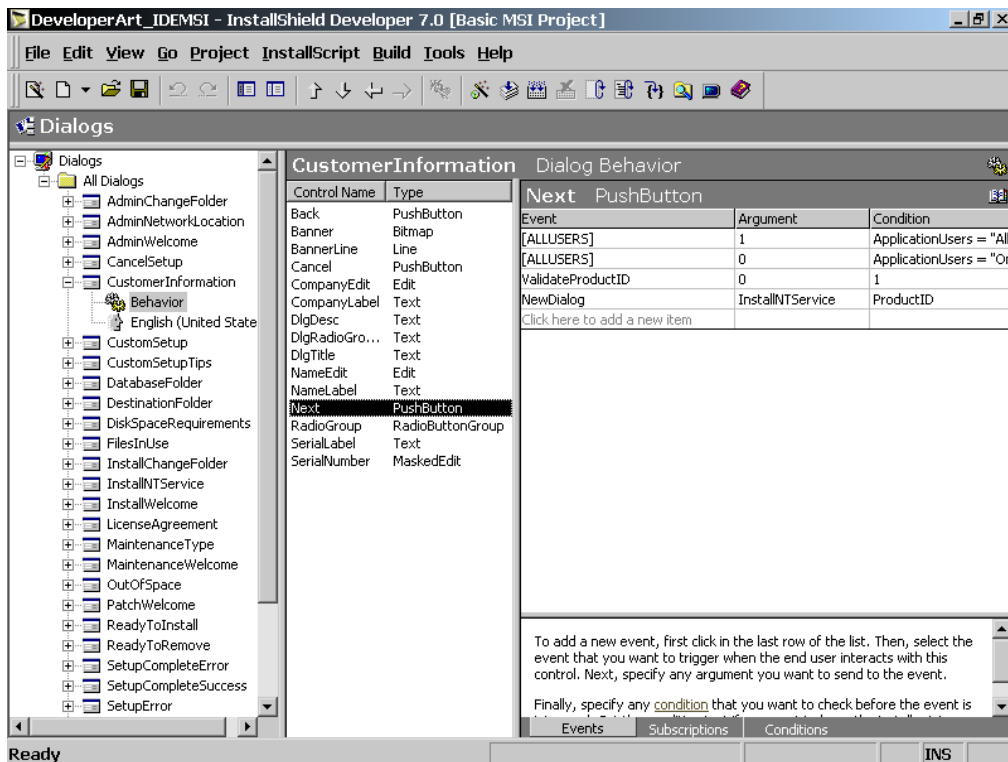


Figure 12-39: The control events for the Next button in the CustomerInformation dialog.

When you make the above changes to the control events for the Next button, you will see what is shown in Figure 12-39.

When you build and run the installation, and click the Next button without entering a serial number, an error message box appears with Error 1701 because you entered the wrong key. When you click OK on this message box, the installation returns to the CustomerInformation dialog where you have another chance to enter a serial number. As long as you enter any characters in the first edit window and any series of numbers in the second and third edit windows of the MaskedEdit control, the serial number will be validated and the installation can continue.

This is not very good security for an installation, but this is all that the Windows Installer offers as built-in functionality. You can use a custom action to validate the serial number input. Then you can more strictly evaluate the validity of the entered value. This is only a little better because a knowledgeable user could easily go into the database using Orca and remove the serial number check. It is because of the open architecture of the Windows Installer technology that Microsoft takes the position that security is the responsibility of the application and not the installation. Accordingly, Microsoft has included certain API functions that allow applications to validate the serial numbers that have been written to the registry. A discussion of these functions is beyond the scope of this book. These functions are fully described in the Windows Installer help file.

Before moving to the next section, you should export the CustomerInformation dialog so that it can be used in other projects. When you export it, you should use a unique file name so that it is clear that it is not the default implementation of this dialog. A file name of CustomerInformationEx_MSI.isd would be a good name to use.

Experimenting with Subscription

Subscription is a functionality that is used to display messages in a progress dialog to inform the user what is happening at a particular point in the installation. To see subscription in action, you will perform a small experiment where you will place a text static control on the SetupProgress dialog and then display certain data in this control while the installation is running.

To perform this experiment, do the following in the Dialog Editor.

1. In the SetupProgress dialog, add a text static control below the progress bar and name this control ActionData.
2. Under the Behavior icon for the SetupProgress dialog, click on the Subscriptions tab at the bottom screen and select the ActionData control. In the Event column, choose the ActionData event from the drop-down menu. In the Attribute column, select the Text attribute from the drop-down menu.

Build the project and run the full installation to see text on the SetupProgress dialog as various actions are executed. Because the Developer Art application is so small, you probably cannot read anything because it flashes by so fast.

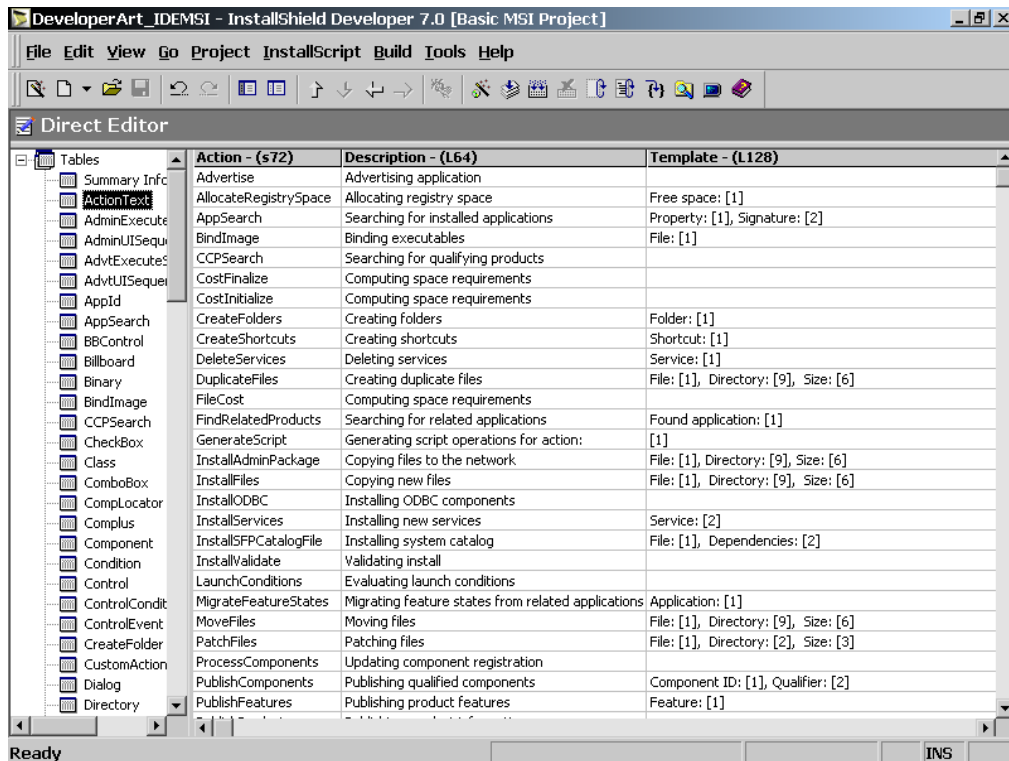


Figure 12-40: The ActionText table is seen from the Direct Editor.

The information displayed in the `ActionData` text control comes from the third column of the `ActionText` table. Go to the Direct Editor view under Advanced Views and click on the `ActionText` table to see what is shown in Figure 12-40.

The first column of the `ActionText` table has the names of standard actions. The second column is entitled `Description` and this column has a description of the type of action being carried out by the action listed in the first column. The third column, entitled `Template`, includes some text as well as numbers inside square brackets. These numbers represent certain data messages that are published by the action listed in the first column.

A text static control that subscribes to the `ActionText` event will display the string in the second column of the `ActionText` table when the action listed in the first column is executed. If you look closely at the design of the `SetupProgress` dialog, you will see that there is already a text static control that subscribes to the `ActionText` control event. When a text static control subscribes to the `ActionData` event, as you just did, the text in the third column is displayed and the numbers inside are filled in with data according to how the action is designed.

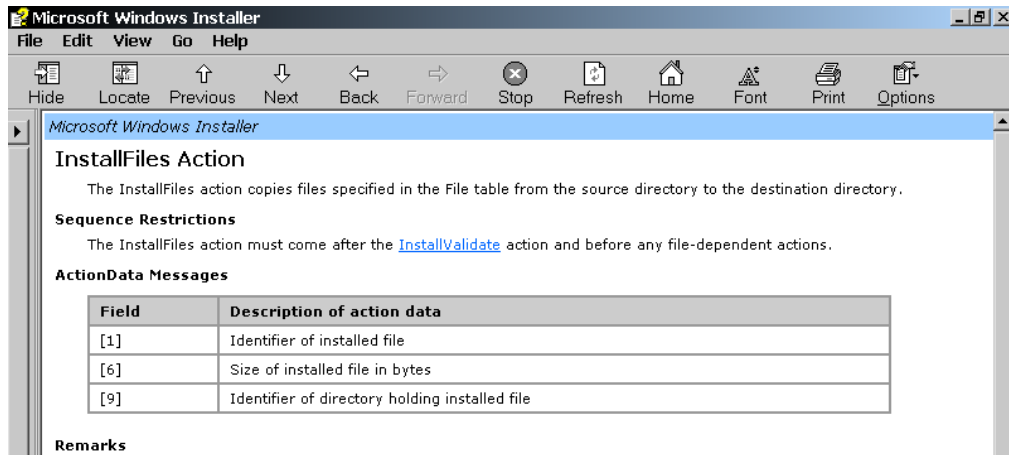


Figure 12-41: *The Windows Installer help topic for the `InstallFiles` standard action.*

As an example, the `InstallFiles` action has the following string in the `Template` column:

```
File: [1], Directory: [9], Size: [6]
```

Look at the help topic for the InstallFiles action in the Windows Installer help file to see what is shown in Figure 12-41.

Figure 12-41 shows a table called ActionData messages. In this table, field [1] gets filled in with the name of the file being copied, field [6] gets filled in with the size in bytes of the file being copied, and field [9] gets filled in with the folder into which the file is being copied. You do not have to display all the data that any action publishes. You can limit the data that is displayed by editing the Template column of the ActionText table. You can also edit the text in the Description column of this table.

Conclusion

This chapter discussed the similarities and differences between how the user interface is implemented in a Standard project versus how it is implemented in a Basic MSI project. It began with a brief overview of how the Windows operating system handles dialogs and then showed that, under the surface, the same actions take place regardless of the project type.

In a Standard project, you define a dialog in a resource-only dynamic link library and then provide an InstallScript function that runs the dialog by responding to messages when the user interacts with the dialog. With a Basic MSI project, you define a dialog in the Windows Installer database tables. The Windows Installer is responsible for reading the database and creating a dialog template in memory. Messages in a Standard project are control events in a Basic MSI project.

In a Standard project, you can use the full range of Windows functionality to define and implement dialogs. The Windows Installer limits you to the controls and messages that are built into the technology. Some controls are not fully functional, as they are in a Standard project. A Standard project provides all the power of Windows programming, but requires a little more effort than what is required in a Basic MSI project.

Chapter

13

Introducing Components

Components were introduced in Chapter 2 and discussed in more detail in Chapter 5. This chapter provides a comprehensive look at components. Components are both the most important part of creating an installation and the most misunderstood part of that installation. It is important to know how various types of components are created, as well as how they are handled by the Windows Installer technology. Microsoft has defined a number of rules for component creation. Understanding the reasons behind these rules can provide insight that will help in the creation of installation packages that are free of problems.

This chapter begins with a detailed look at Microsoft's component rules and then it provides some examples of what can happen when these rules are not followed. After discussing these rules and the ramifications of breaking them, you will create different

types of components, in this chapter and the next, using the tools that are available in InstallShield Developer.

Components and the Windows Installer

The subject of how the Windows Installer handles components is important because, regardless of the project type you use to create an installation, the Windows Installer handles the actual modifications to the target system. When the Windows Installer makes changes to a system, it does so by adding components or removing components from the system. The Windows Installer treats a component as an indivisible entity so it is all or nothing when a component is added to a system during an installation. Before discussing the way in which the Windows Installer handles components, we need to look at a few definitions.

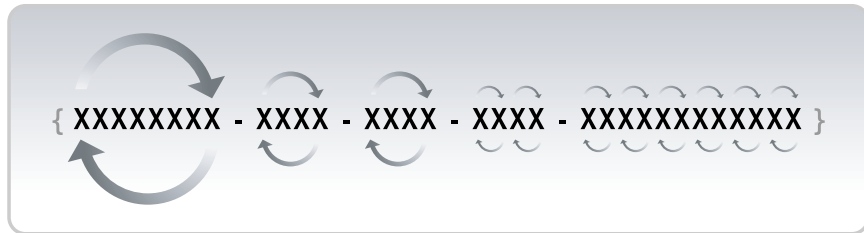
Some Definitions

The Windows Installer keeps track of all components that are installed. It does this by using the registry. You will be looking at the registry a lot to see what the Windows Installer is doing there. To understand what you are seeing, you need to first define a few of the information types that the Windows Installer writes to the registry. These definitions are listed below:

Packed GUID: The definition of a packed GUID is the most important one to understand because it will allow you to look in the registry and understand what is happening when the Windows Installer registers a component. The standard presentation of a GUID that you see in your projects is the one that requires 38 characters. A packed GUID is a representation that requires only 32 characters and the order of the characters is modified to enhance registry searches.

There are two actions that you need to take to convert a standard GUID into a Packed GUID.

- First, reorder the hexadecimal characters as shown in the following diagram.



The first group of eight and the first two groups of four hexadecimal characters are written in reverse order. Starting with third group of hexadecimal characters, every two characters are switched.

- After you have reordered the hexadecimal characters in the GUID, remove the curly braces at the end of the GUID and also remove the separating dashes, thus creating a Packed GUID.

As an example, look at the component code for the DeveloperArt component as it exists in the DeveloperArt_IDEMSI.ism project on the included CD-ROM and note how the Packed GUID differs from the component itself.

```
{45858C51-669F-4ACB-8310-26102F56F3F3} - standard format
```

```
15C85854F966BCA438016201F2653F3F - packed format
```

Compressed GUID: A compressed GUID is another representation that the Windows Installer uses to further reduce the space in the registry to write a GUID. A Compressed GUID requires only 23 characters and is used primarily to construct a *Darwin Descriptor* (see description below). The following is an example of a Compressed GUID:

```
10!!!gxsf(Ng]qF`H{Ls
```

An understanding of this format is important so you know what you are looking at when you see a jumble of characters like this in the registry. The particular Compressed GUID shown above is part of a Darwin Descriptor that was written to the registry when Microsoft Office 2000 was installed.

Darwin Descriptor: A Darwin Descriptor is a combination of the product code GUID, a feature name, and a component code GUID. There are four different representations used by a Darwin Descriptor depending on the number of

features and components that a product contains. The most common representation is as follows:

```
{compressed product code}{feature name}>{compressed component ID}
```

In the above representation, the curly braces are there only as separators and they do not appear in the actual Darwin Descriptor. An actual Darwin Descriptor from the registry is as follows:

```
10!!!gxsf(Ng]qF`H{LsOfficeWebComponents>QLw$@8Dmf(y~S~pL"F`
```

Darwin Descriptors can be combined into a list in the registry by using the NULL character as a separator. When more than one Darwin Descriptor is used as the data for a value name in the registry, it is called a Darwin Descriptor List. The word Darwin is the code name that was used by Microsoft at the inception of the Windows Installer technology.

Keeping Track of Components

When the Windows Installer installs a component, it keeps track of where it was installed and which product installed it. The concept of the Packed GUID is important for gaining an understanding of how the Windows Installer registers a component when the component is installed either locally or run from source. If you look in the registry in the following location, you can start to see how components are registered.

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\  
Installer\UserData\
```

In this location, the UserData key has a number of sub-keys (Figure 13-1). Directly under the UserData key are keys that are named using the security identifier (SID) for each user who has an account on the machine. This applies to machines running Windows NT 4.0, Windows 2000, or Windows XP. Under each key that is named using the value of a user's SID, there are three keys named Components, Patches, and Products respectively. In this discussion it is the Components key that is important.

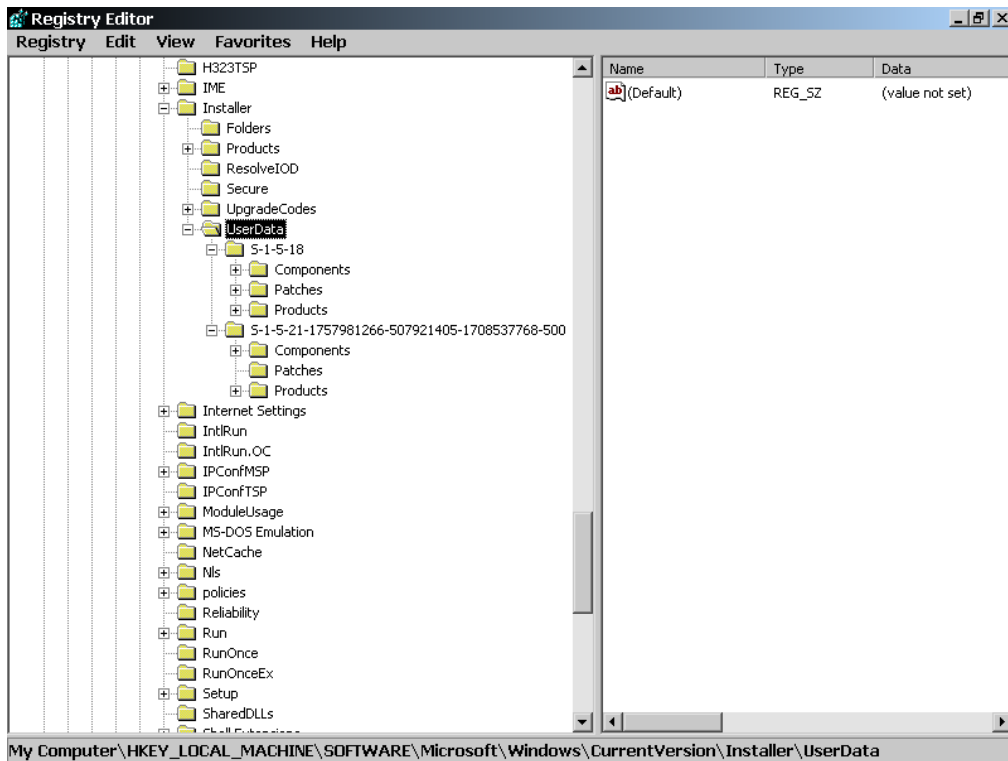


Figure 13-1: *The registry location where the Windows Installer registers components.*

The SID S-1-5-18 goes by the name LocalSystem and is a special account, which has all the access that an administrator has. When you are signed on to the target machine as the administrator, the components of products that you install are registered under this key. Expand the tree under the Components key and you will see a long list of keys that have numbers as their names. Each key represents a component that has been installed on the target machine. The name of the key is the Packed GUID representation of the component code. The value or values that are written against each of these component keys is a value name that is the Packed GUID representation of the product code of the product that installed the component. The value data that is written against the value name is the key path of the component.

While browsing through the list of registered components on my machine, I was able to find an entry that illustrates several important points about the registration of components. This component is shown in Figure 13-2.

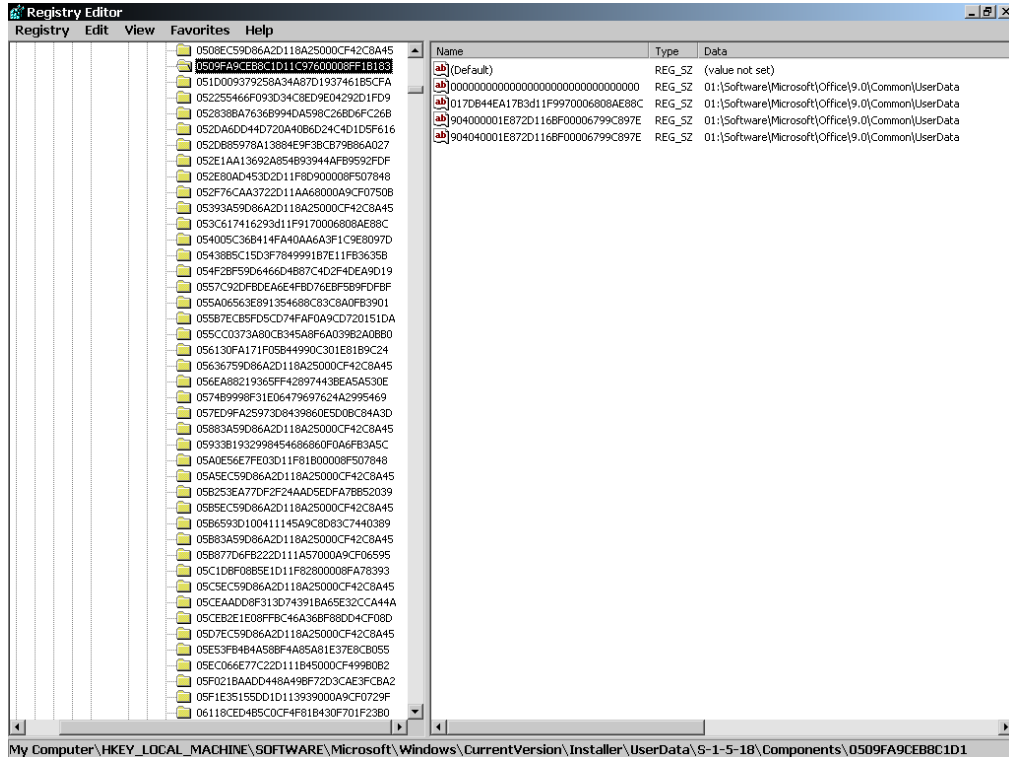


Figure 13-2: *Example of a component installed by more than one product.*

The values written against the component shown in Figure 13-2 tell us that three different products installed this same component. Every time that one of the products that installed this component is uninstalled, the value that represents that product is removed from the registry until all products are uninstalled. When all products are uninstalled, the Windows Installer removes the component from the system. This is the mechanism used by the Windows Installer to reference count components. Reference counting is used to prevent the uninstallation of one product from disabling another product that is still on the machine. Reference counting is critical for any component that is shared between products or between features of the same product.

Figure 13-2 also tells us that this component will never be removed from the system because one or more of the products that installed it have made this component permanent. A component is made permanent by writing into the one of the values a dummy product code that consists of all zeros. Since there will not be a product that has such a product code, there will never be a product that can remove this last value from the registry. This means that the Windows Installer will never remove the component from the system.

The value data for the component shown in Figure 13-2 indicates that the key path for this component is a registry entry. The 01 at the beginning means that the registry entry is written under HKEY_CURRENT_USER and that the key path is the value name UserData written against the Common registry key. There are a number of different character combinations that are used to indicate different types of key paths for a component. It is important to understand that this registry entry identifies the component's key path.

Regardless of whether the end user performs a per-machine or a per-user installation, components are always reference counted in this particular location under HKEY_LOCAL_MACHINE. This is why any installation using the Windows Installer requires that the end user have administrative privileges or that the LAN administrator has granted elevated privileges. There are other component related keys in the registry but they are used to handle qualified components, not to perform reference counting. qualified components are discussed briefly at the end of this chapter.

The Component's Composition

A component can be composed of anything that you want to add to the target system. A component can also contain the logic for controlling items already on the target system, such as an NT service. A component contains resources, which might be files, shortcuts, registry entries, and any other items that can be added to a machine.

Because a component is installed as one unit and removed as one unit, it is important that the resources you place in a component are related to each other. A component is identified by the component code that is assigned. Two components that have the same component code are considered the same entity even if they contain different resources. The Windows Installer handles the situation just described in two different

fashions depending on whether the key path for the component is a versioned file or not. If the key path is a versioned file, the version of the file defines the version of the component. In this case, a component that is already on the system will not be updated with new resources unless the modified component that is being installed has a key path file with a greater version. If the component has a key path that is not a versioned file or is a folder or registry entry, the modified component modifies the target system with any new resources and uses the file versioning rules to decide whether to replace files already on the system. The file versioning rules are discussed in detail in the Windows Installer help.

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

Creating components that do not conform with how the Windows Installer handles components can result in situations that range from not being able to completely uninstall an application to disabling applications that are already on the system. The next section looks at how each of these situations could arise.

Leaving Resources Behind During Uninstallation

For this discussion, we will assume that there are two products, where the first product installs the original version of the component and the second product installs a newer version of the same component where resources have been added. The first product is installed and then the second product with the modified component is installed. At this stage, the component has a reference count of two in the registry, as described in the Keeping Track of Components section. The second product is uninstalled and the only thing that happens is that the reference count is decremented by one. When the first product is uninstalled, the cached installation database for this product does not have knowledge of the new resources that the modified component

added to the system. The resources that were added by the installation of the modified component are left behind when the first product is uninstalled.

A similar situation arises when the modification of an existing component consists of removing resources instead of adding them. In this scenario, the order in which the two products are uninstalled is reversed. Here the first product is uninstalled first and then the second product. Once again, resources are left on the system.

Disabling Installed Applications

This section presents two completely different scenarios, both of which result in disabling an installed application. In the first scenario, a component is modified by adding a new file that is incompatible with the installed product's requirements. Another product installs this component and the reference count is incremented to two. Since the new file has a later version than the file in the original version of the component, the file that on the system is overwritten. The product that requires the original version of the file is disabled because it cannot use the new version of the file.

In the second scenario, there are two different components with two different component codes. The two different components install the same resource. The two products that contain these components are installed and each of these components gets a reference count of one. One of these components is removed when the product that installed it is uninstalled. The reference count is decremented to zero and the shared resource is removed, thus disabling the other product that is still installed and needs the resource.

Determining When to Change the Component Code

It is okay to keep the same component code when revising an existing component when testing shows that the revised component is completely backward compatible with all previous versions of the component. This might be possible for the simplest of components, but it can be argued that true backward compatibility is nonexistent and that any change to a component requires that a new component code be assigned.

If you agree with the supposition that true backward compatibility does not exist, you have to do some additional work when revising a component, in addition to changing its component code. You have to change the name of every resource that is being

installed by the new component so that it does not conflict with the resources installed by the previous component. This is necessary to avoid overwriting the resources installed by the previous component. These required changes include the following:

Files: Change the name of the files in the component or change the name of the folder into which the file is to be installed.

Registry Entries: Change the name of the key under which values and data are written.

Shortcuts: Change the name of the shortcut.

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

The best way to understand the impact of ignoring the basic requirements for constructing a component is to run experiments with two different products that install the same components. There are two such projects on the included CD-ROM that install two different versions of the same simple application. The names of these two projects are Editor10.ism and Editor20.ism. You can add resources to one version of the component and see what happens when both products are installed.

Rules for Creating Components

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

This section reviews the guidelines that Microsoft has defined for creating components. After discussing the Microsoft guidelines, this chapter provides some guidelines for the remaining files that are typically part of an application.

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

The above rules provide direction for only a subset of all the files that normally make up an application. For the remaining files of an application, you need to make some decisions before you can know how to place these files into components. The first decision that needs to be made is to assess whether there are any resources that might be acceptable now to ship in the same component but are likely to be shipped separately in the future. If you can make that determination, you should ship these resources in separate components now and not wait until later.

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

Now that we have looked at the general rules for creating components, we need to look at the guidelines that define specific actions required when creating components. InstallShield Developer handles some of these rules when you author a component, but you are responsible for compliance with other rules. These rules are discussed in the following list:

- Depending on the type of key path that is defined for a component, there need to be reciprocal relationships between the Component, File, and Registry tables. The Windows Installer uses the key path defined in the Component table to detect the presence of the component on the target system during an installation. The key path is normally a foreign key into the File or Registry table and both tables have a reference back to the Component table.
- A component that installs font files needs to have as its destination, the location defined by the FontsFolder path property. The Windows Installer sets the value of this property to the absolute path of the target system Fonts folder. When a font component is created, an entry must be made in the Font table. The Component Wizard in InstallShield Developer makes the proper entries when you use it to create a font component. One thing that is necessary to do for a font component that is not handled by the Component Wizard is to make the font component permanent. Fonts are not reference counted so unless you make a font component permanent it will get uninstalled when the component is uninstalled. This may be what you want if the font is proprietary to your company.

- There should never be two different components created that have the same component code. Since the component code is not part of the primary key, InstallShield Developer builds the Component table into the database with the duplicate component code. No warning is issued.
- A component that has a destination set to the SystemFolder property should be defined as a permanent component by setting the Permanent property to Yes for the component in the Setup Design view.
- A component that uses an empty folder as the key path for the component needs to have the empty folder defined in the CreateFolder table.
- To enable advertisement, any component that installs the extension server for an extension listed in the Extension table or is referenced by an advertisable shortcut listed in the Shortcut table must have a component code.
- A dynamic link library that is called using the load-time linking method must be placed in a component associated with the feature that installs the DLL's client.
- If a component contains a file that is on the System File Protection list, that file must be the key path for the component. This means that there needs to be a separate component for every file included in the installation that appears on the System File Protection list. This rule is not automatically covered by the other rule about .exe, .dll, and .ocx files since there are many more files with different extensions that appear on the System File Protection list. Also, a file that is on the System File Protection list needs to be marked as permanent and needs to be installed locally and it cannot be installed to run from source.
- Every component that is defined in an installation must be associated with a feature. InstallShield Developer handles this requirement for you.
- There cannot be two components that install the same named file to the same folder. This points out that the name of a file resource consists of the absolute path of the file plus the name of the file and its extension. As long as two files with the same name are installed to two different locations, there is no danger of overwriting a resource of one component with the resource of another component.

- A component that installs a COM server or an extension server should not use the Registry table to make the necessary entries in the registry at installation time. Instead, the Class, Extension, ProgId, Verb, and MIME tables should be used. This is necessary to support advertisement because the entries in the Registry table are not written until the associated component is installed. The entries listed in the Class, Extension, ProgId, Verb, and MIME tables are written at the time the application is advertised. InstallShield Developer handles this in a correct manner.
- A component that is defined to run from source should not be compressed into a cabinet file.
- Any component that installs to a user's profile must have a key path that is a registry entry written to HKEY_CURRENT_USER.
- Any component that installs a standard shortcut must have a key path that that is a registry entry written to HKEY_CURRENT_USER.
- Any component that creates registry entries of the REG_MULTI_SZ, REG_BINARY, REG_EXPAND_SZ, or REG_DWORD needs to be conditioned so that it will not be installed on Windows 95. The Windows 95 registry does not support these types of entries. It supports only the REG_SZ type.
- A component should not use a companion file as the key path. Companion files are discussed at the end of this chapter.
- A component must not include both per-machine and per-user data. For example, this means that a component should not write registry entries to both HKEY_LOCAL_MACHINE and HKEY_CURRENT_USER. If it is not known until installation time whether it will be a per-machine or a per-user type of install, all registry entries need to be defined under the HKEY_USER_SELECTABLE key in InstallShield Developer.
- A component should not install an application file, .ini file, or a shortcut file into a per-user only folder unless the package is designed specifically to perform only a per-user installation. The predefined per-user-only folders are: AppDataFolder, FavoritesFolder, NetHoodFolder, PersonalFolder,

PrintHoodFolder, RecentFolder, SendToFolder, MyPicturesFolder, and LocalAppDataFolder.

- If a component installs a non-advertised shortcut, the shortcut's target must be installed by the same component. This rule is to avoid having a component that installs the shortcut or the component that installs the shortcut's target from getting into different states.
- A component that has a NULL component code must not be defined as being a permanent component.
- Do not specify a component to be run-from-source if this component is installing an NT service. The Windows Installer does not support this type of installation and any NT service that runs with the privilege level of the local system must be run from the local hard drive.

Now that you know the rules that you should follow when creating components, we can discuss the component creation tools available in InstallShield Developer. After that, you will be able to create some components.

The Component Creation Tools

So far in this book, we have discussed how components are automatically created when you add files to a feature in the Features view. You have also created components in the Setup Design view under Advanced Views. Each of these approaches is acceptable for creating a few components that install one or several files that do not require any special treatment beyond that discussed in Chapter 10.

However, applications usually need to define components that contain many files, the number of which fluctuate dramatically during the development process. There are also a number of special components that need to be created and that have unique properties such as an NT service component or an ODBC component. This section introduces the InstallShield Developer tools available for creating special types of components.

Dynamic File Linking

Most applications have at least a few components that install a lot of files, such as graphics files or templates. The number and types of these files can change dramatically from build to build, so it can require a substantial effort to keep the files up-to-date that have been linked into this type of component. To solve this problem, InstallShield Developer provides the capability to link to a folder, called a dynamic link, and to use wild cards to specify which files in the folder should be included in the component. When you link to individual files, you are creating a static link. It is a time-consuming manual operation to remove links to files that no longer exist in the component and to add static links to new files, particularly if this needs to be done before every build.

Dynamic File Linking Basics

You can initiate dynamic file linking for a component by doing the following:

1. Go to either the Setup Design view or the Components view under Advanced Views.

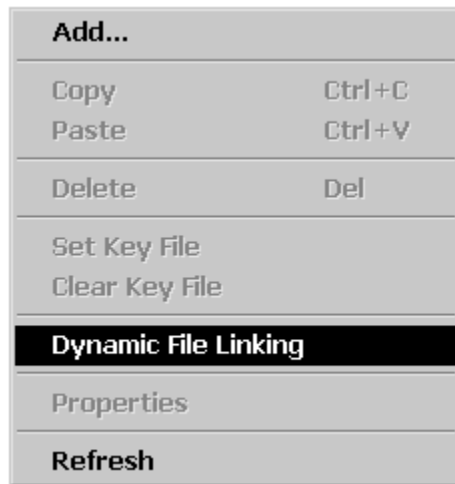


Figure 13-3: *Context menu for initiating dynamic file linking.*

2. Click on the Files icon under the name of the component where you want to implement dynamic file linking.
3. Right-click in the panel to the right of the screen that lists the component's files and select Dynamic File Linking from the context menu (Figure 13-3). The Modify Dynamic Links dialog is displayed (Figure 13-4).

Using the Modify Dynamic Links dialog, you can create new links, modify links, and delete links. If no links exist, only the New Link button is enabled. If one or more links already exist in the component and you click on one of those links, all three buttons are enabled.



Figure 13-4: *The Modify Dynamic Links dialog showing one dynamic link.*

There are four columns of information in the Modify Dynamic Links dialog. This information displays, in the first column, the name of the path variable that points at the folder to which the link has been created. The other three columns show the options that were selected when the dynamic file link was created.

To create a dynamic link do the following:

1. Click the New Link button to display the Dynamic File Link Settings dialog (Figure 13-5).
2. In the Dynamic File Link Settings dialog, browse to the folder where the files to be included in the component are located.

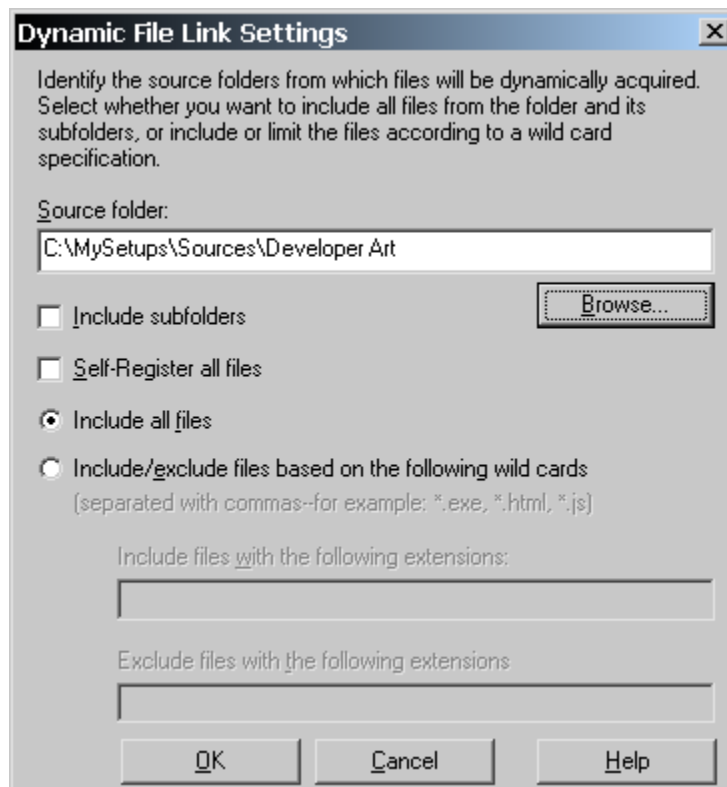


Figure 13-5: *The Dynamic File Link Settings dialog.*

3. Select the options that you want. The impact of selecting one or more of the options in the Dynamic File Link Settings dialog is discussed in the next section.
4. Click OK to close the Dynamic File Link Settings dialog and display the Path Variable Recommendation dialog.
5. Select or enter the path variable to be used to represent the dynamically linked folder and click OK to close the Path Variable Recommendation dialog.
6. In the Modify Dynamic Links dialog, complete the dynamic file linking operation by clicking OK or click the Apply button to create a dynamic link to another folder for the same component.

As an experiment you can open the DeveloperArt_IDEMSI.ism project and add a temporary feature and component. For the component, create a dynamic link to the folder where the source files are kept for the Developer Art application. When you create this dynamic link, you will see something like what is shown in Figure 13-6.

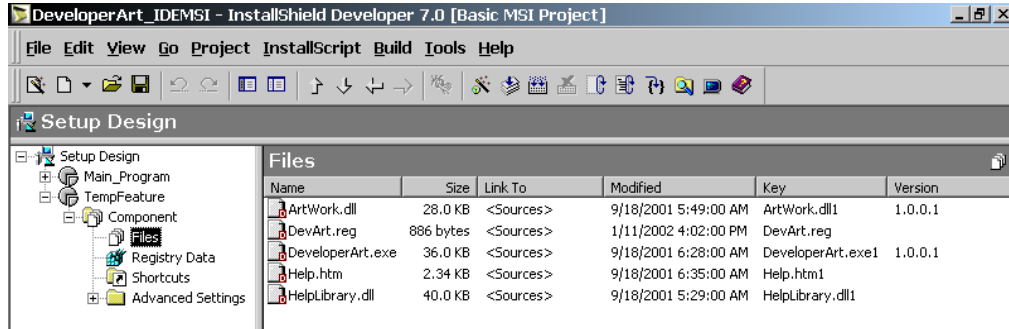


Figure 13-6: A dynamic file link to the Developer Art source folder.

There are several interesting things to note in the Component Files view (Figure 13-6). If you right click on one of the files in the list, you will see that you cannot select a dynamically linked file as a key path for the component. This is because the component sees only the folder and the files that are in the folder are added to the project only at build time. The files in the folder can change over time and a file could be removed from the folder between builds. If you want to designate a file as the key path for the component, then that particular file needs to be statically linked. In

addition, if the file resides in the folder at which the dynamic link points, it needs to be specifically excluded from the dynamic link using the exclude option in the Dynamic File Link Settings dialog.

Another thing to note in the Component Files view (Figure 13-6) is that dynamic file linking does not recognize the best practices rules that were discussed in Chapter 5. When the appropriate option is selected, InstallShield Developer enforces these basic rules. Dynamic file linking is used to include files from a folder, so the dynamic link itself contains no intelligence about the file extension. Because of this, it is not possible to enforce best practices rules for files included using dynamic file linking.

To make dynamic file linking work efficiently, you need to create a specific directory structure on your build machine that matches the directory structure of your application after it is installed. The directory structure should match all the way down to the leaf folders in the directory tree. Where dynamic file linking is applicable, you can create links to these folders on the build machine and, before each build, you can populate each folder with the latest files to be included in the application. This process can be more efficient with the dynamic linking options discussed in the next section.

Dynamic File Linking Options

The Dynamic File Link Settings dialog (Figure 13-5) offers a number of options. The “Include subfolders” option allows you to include any subfolders that may be under the folder identified in the “Source folder” edit field. The “Self-Register all files” option allows you to have all the dynamically linked files self-registered. The subject of self-registration is covered in the COM Components section in Chapter 14.

The “Include all files” and “Include/Exclude...” radio button group allows you to either include all files in the source folder, the default, or to include and/or exclude files based on file specifications that can use wild cards if necessary. Multiple specifications are separated using a comma delimiter. There is also a method that can be used to maintain the consistency of component codes and file identifiers used from build to build.

INCLUDING/EXCLUDING FILES

To include or exclude specific file types from your component, select the “Include/exclude files based on the following wild cards” option and enter the

specifications for files that you want to include and/or exclude from the component. If you select the second radio button and enter nothing in either of the edit fields at the bottom of the Dynamic File Link Settings dialog, the component will include all the files in the source folder.

One of the most common uses for excluding files from the dynamic link is to identify a file that should always be used as the component's key path. When you do this, you can add the excluded file statically and then set it as the key file. As an exercise, go to the component shown in Figure 13-6 and make the `ArtWork.dll` file the component's key file. To do this:

1. Right-click in the Files list panel and select Add.
2. Browse to the location where the `ArtWork.dll` file is located and add this file to the component.
3. The Path Variables Recommendation dialog appears. Make your selections in the dialog and click OK. The following dialog appears (Figure 13-7).

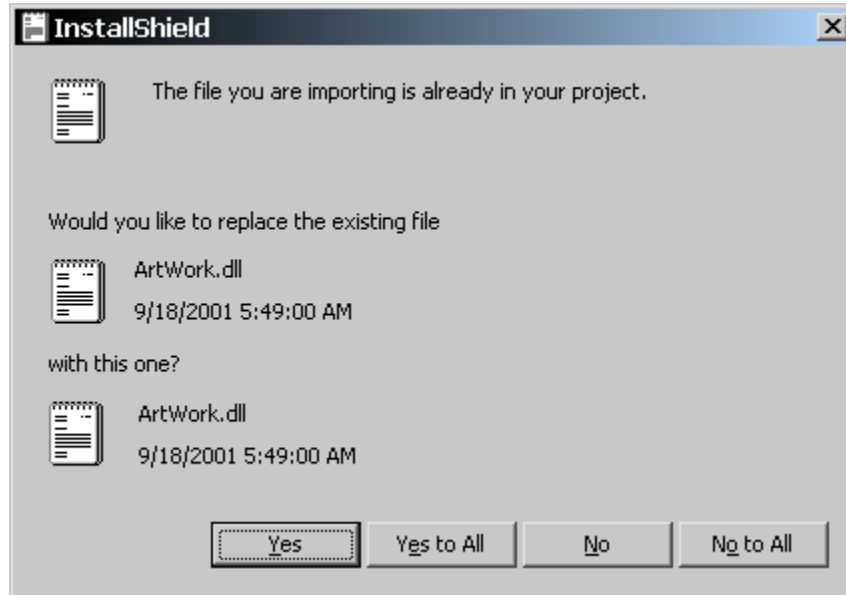


Figure 13-7: *The file replacement dialog.*

4. Click Yes to statically add the ArtWork.dll file to the component instead of adding it dynamically. Note that the icon beside this file has changed and that you can now set it as the component's key path.
5. Set this file as the key path by right-clicking on the file and selecting Set Key File. A yellow key icon appears beside this file, indicating it as the component's key path.

By performing the actions described in the previous procedure, you have implemented changes in the dynamic file link specification. To view what happened to this specification, go back to the Modify Dynamic Links dialog, select the dynamic link and click Modify. You should see what is shown in Figure 13-8.

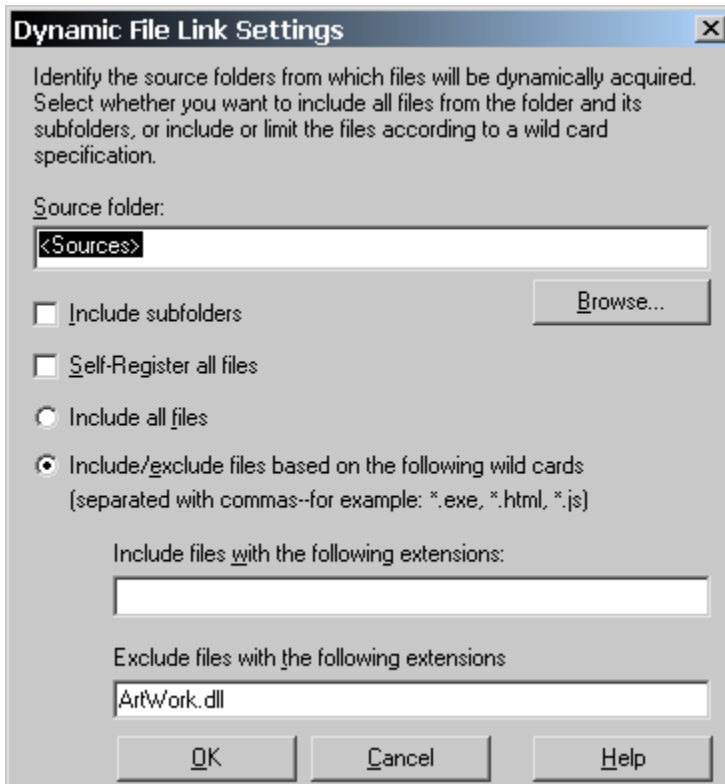


Figure 13-8: *The modified dynamic link specification.*

The `ArtWork.dll` file name appears in the Exclude files edit field and the “Include/exclude...” radio button is selected. Deciding on the file or files to be included/excluded prior to setting the dynamic link would be the proper approach to follow. Also, one of the first actions that you should take after you create a component is to statically link the file that is to be the component’s key path and set it as the key path.

INCLUDING SUBFOLDERS

When you select the “Include subfolders” check box in the Dynamic File Link Settings dialog, you enable the creation of additional components that contain the files that exist in the subfolders. These additional components are created only at build time and, as such, are not visible in either the Setup Design or Components views. However, the files that are contained in these subfolders and therefore included in the additional components are displayed in the Files list panel.

To understand how including subfolders works, do the following example:

1. Create a subfolder named `SubFolder` under the folder where the Developer Art source files are stored.
2. Copy into this subfolder the all the Developer Art source files so you now have a parent folder and a subfolder that contain the same files.
3. Go to the component that is shown in Figure 13-6, except now it has the `ArtWork.dll` file as the key path for the component.
4. Modify the dynamic link specification for this component to include subfolders. The result of these actions is shown in Figure 13-9.

You can see in Figure 13-9 that there are no new components shown on the left of the screen but that there are more files displayed in the Files list panel. The new files now include the subfolder as part of the file name. When subfolders are added to the dynamic file link specification, there is a default limit of 500 files from all subfolders that can be shown in the Files list. This is to avoid the situation where the subfolders that are part of the dynamic file link contain thousands of files. Modifying the data for the `FileItemsCount` value under the following registry key changes the number of files that can appear in the Files list.

HKEY_CURRENT_USER\Software\InstallShield\Developer\7.0\

Project Settings

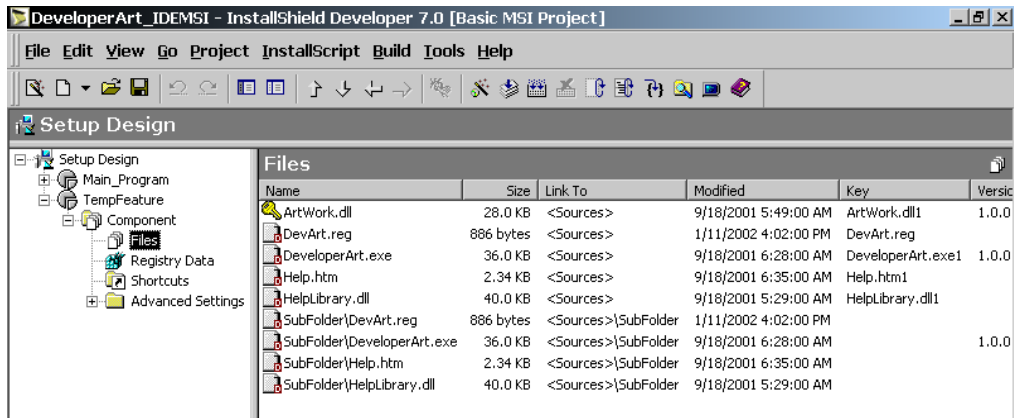


Figure 13-9: *The dynamic link that now includes subfolders.*

You need to think of the component where the dynamic link was first created as the parent folder. All components that are created because you selected the “Include subfolders” check box will inherit all the properties of the parent component, except the destination property and the key path property. The destination property for each of the child components is created from the destination property for the parent component. Appending the path to the subfolder on the build machine modifies the destination for the parent component. This modified path becomes the value of the destination property for the child component.

For the example shown in Figure 13-9, the component used to create the dynamic file link has a destination defined by the `INSTALLDIR` property. The folder into which the files in the subfolder will be installed on the target system will be the value of the `INSTALLDIR` property with a subfolder under this location named `SubFolder`. The default for this location will be the following if the end user does not change the destination during the installation.

```
C:\Program Files\InstallShield\Developer Art\SubFolder
```

The key path for each of the components that is generated from a subfolder on the build machine is the first file in an alphabetically ordered list of the files in the component. The key path for the components that are dynamically created at build time from subfolders cannot be modified inside the project file. Only post-processing

the Windows Installer database after it is created can do this. When you build the project with the temporary feature and component as shown in Figure 13-9, four warnings appear to indicate that different components are installing the same four files to the same location. To eliminate these warnings, go to the Destination property for the temporary component shown in Figure 13-9 and add a subfolder name under the [INSTALLDIR] property. For example, modify the Destination property as follows:

```
[INSTALLDIR] \SubFolder
```

When you make this modification, the files in the parent component will be installed to a folder named `SubFolder` under the root install location for the product. The files in the component created from the subfolder named `SubFolder` on the build machine will be installed to a folder under the root install location of `SubFolder\SubFolder`. After modifying the Destination property, the build will not generate any errors. The media image for this build looks like the default installation directory structure (Figure 13-10).

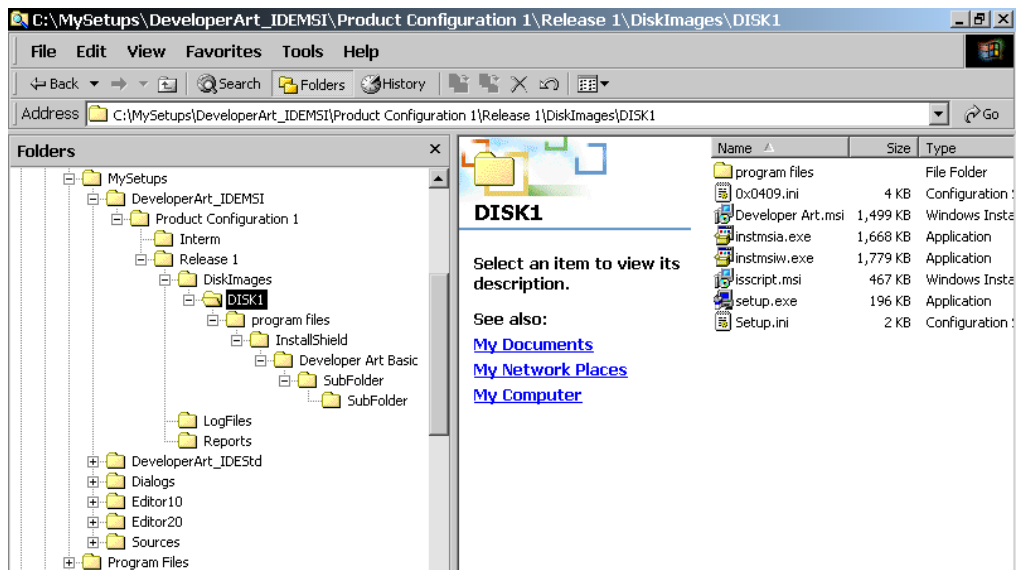


Figure 13-10: *The media directory structure for components created using dynamic linking.*

In this section we have been using terms such as parent component and child component. Note that this is only a build-time construct. In the Windows Installer

world, there is no such thing as a subcomponent. All components are on the same level and it is only features that can have a hierarchy.

MAINTAINING CONSISTENCY BETWEEN BUILDS

When subfolders are included in the dynamic file linking settings, each subfolder defines a separate component, as discussed in the previous section. Each time the project is built, the components generated from the subfolders are recreated and this generates new component codes and other identifiers as necessary to link the various tables together. The dynamically generated component codes and other identifiers are all GUIDs, with most of the identifiers a modified version of a GUID. You can see this by opening up the `DeveloperArt_IDEMSLism` file created in the last section. This file can be opened up using Orca and Figure 13-11 shows a portion of the Component table for this build.

The highlighted row in Figure 13-11 is the dynamically created component. The component identifier in the first column is created from the component code in the second column. The component identifier modifies the component code by removing the dashes and the curly braces and adding an underscore character in front. All other identifiers are also GUIDs that are modified in the same fashion. An immediate example of this is the entry in the `Directory_` column where there is a foreign key into the `Directory` table created from a modified GUID generated during the build. Browse through this database using Orca to see all the locations where dynamically created entries are entered into tables.

Component	ComponentId	Directory_	Attr
Component	{72A6FFC8-2132-492B-9282-E2E703F98EDB}	SUBFOLDER	8
Condition	{45858C51-669F-4ACB-8310-26102F56F3F3}	INSTALLDIR	8
Control	{0680DAE5-3730-432F-819C-F3312EE0225F}	INSTALLDIR	8
ControlCo...	{F167E3CB-3EF6-49B7-870E-01C6F222FCF0}	INSTALLDIR	8
ControlEv...	{8157BDDA-C136-422B-AC89-602758CE09C2}	INSTALLDIR	8
CreateFol...	{59C8168F-C16F-49D6-B147-FED7A395BAA5}	INSTALLDIR	8
CustomAc...	{08B5838-682D-42D8-A249-92DBAA007791}	INSTALLDIR	8
Dialog	{D38B0B5E-0F9D-47F3-BBC9-CE53B7500155}	INSTALLDIR	8
Directory	{F49CE77B-098D-482E-BA5A-AC8B3C71B706}	._A9736B89F6D14DBBB29667228E6E5C5F	8
DrLocator			

Figure 13-11: *The Component table for a build that contains a dynamically created component.*

There will be situations when you want to know the value of these dynamically created identifiers. You can use this information for performing post processing on

the database after a build or you might want to have a custom action that can access one of these components during the installation. To do this, you need to have some method for preventing these component codes and identifiers changing from one build to another. You can do this using the Previous Package property under Step 7 in the Releases view.

The Releases view displays a tree of all the product configurations and releases that have been generated for a particular project. One of the properties that can be set for a release is the Previous Package property. The documented purpose of this property is to allow the optimization of patch packages. However, identifying a Windows Installer package in this property also serves the purpose of forcing each build to use the same component codes and other identifiers that were generated for the dynamically created components in the referenced package.

To experiment with this, follow the steps listed below:

1. Build the project that contains the components using dynamic file linking and are also using subfolders to create dynamic components.
2. After each build, open the Windows Installer database in Orca and verify that the component code and other GUID based identifiers change each time.
3. Copy the DeveloperArt_IDEMSI.ism file to a location that is not in the path of the build. A good location would be a folder directly under the MySetups folder. You need to do this because every complete build deletes the complete build directory structure and recreates it.
4. Identify the file that you just copied as the value for the Previous Package to be used for the release. You do this in the Releases view as shown in Figure 13-12.
5. Run several more builds and, after each build, open the Windows Installer package using Orca. Note that the component code and identifiers no longer change from one build to the next.

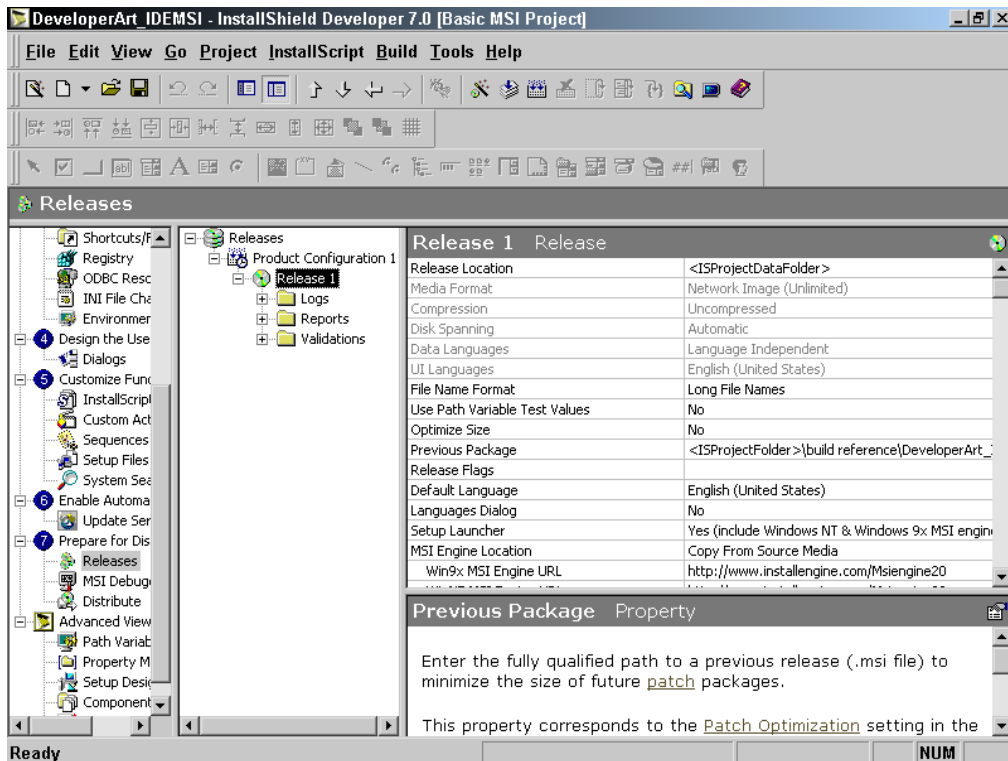


Figure 13-12: *Specifying a base MSI package in the Releases view.*

InstallShield Developer adds a custom table named ISDFLInfo to the Windows Installer database whenever there are dynamically created components. The information in this table enables the build process to use a previous package as the basis for maintaining the same component code and identifiers for each dynamically created component. Whenever your project adds new subfolders so that new dynamic components are created, you need to copy the first build of the Windows Installer package to the location where the previous packages are being stored. This way, you maintain consistency of the component codes and identifiers from one build to the next.

Creating components that use dynamic file linking is a great efficiency booster. However, because there is no real control over components that are created at build time, dynamic file linking is valid only for creating components that copy many files to the target system and when these files do not need associated registry entries, shortcuts, or environment variables. When creating components that have all of

these other requirements, these components must be created statically so each component's associated data can be added into the project. InstallShield Developer provides a tool called the Component Wizard to assist you in creating components.

The Component Wizard

You can access the Component Wizard from the Setup Design view, the Components view, or the Files view. The components created with this wizard are added to the project file and can be edited as necessary. In the Setup Design view, you can launch the Component Wizard by right clicking on a feature in the Setup Design tree and selecting Component Wizard. In the Components view, you can right-click on the Components tree icon and select Component Wizard.

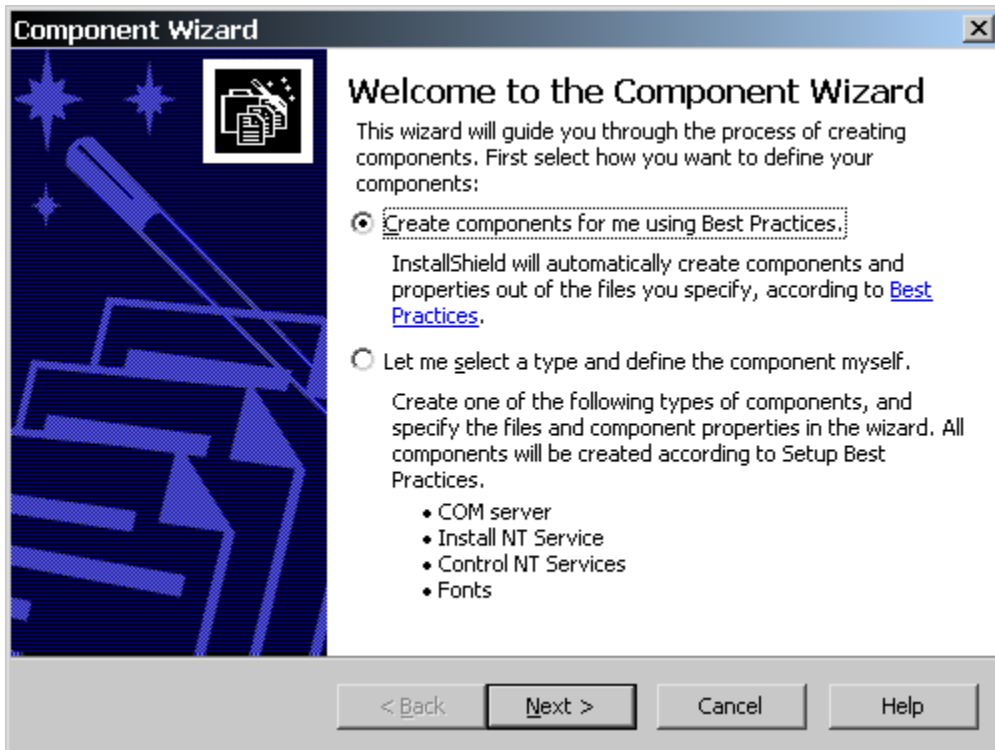


Figure 13-13: *The Component Wizard Welcome panel.*

Even though the same Component Wizard is launched from any of these views, the results are a little different. In the Setup Design view, the Component Wizard creates a component that is associated with a specific feature. If you want to associate the component with other features, you have to go to each of the other features, right click on the feature, and select the Insert Components option to add the component to the selected feature. In the Components view, the Component Wizard creates a component that is added to the project file but is not associated with any feature. When a component is not associated with a feature, it will not be built into the Windows Installer database. To associate a component created in the Components view, you need to switch to the Setup Design view and use the Insert Components option to add the component to one or more of the features.

When the Component Wizard is launched, an initial panel provides you two different approaches that can be used to create components (Figure 13-13).

In the Welcome panel, you can select to create components automatically using the Best Practices rules or to create individual components of special types. We will discuss each of the special types in Chapter 14. This section discusses what happens when you want to automatically create components using the Best Practices rules.

In the first part of this chapter, we looked at many rules for creating components. The Best Practices rules as defined in InstallShield Developer are a small but important subset of all the rules. The Best Practices rules are repeated here in the following list:

- No component should be created to install a file if that file can be installed by a component already contained in a merge module.
- Every .exe, .dll, .ocx, .hlp, and .chm file needs to be in its own component.
- Each of the file types listed in the second rule needs to be set as the key path for its component.

The process that the Component Wizard follows to create a set of components is to first ask for the destination to be used for all the components that will be created. The input of the destination folder is done in the Destination panel (Figure 13-14).

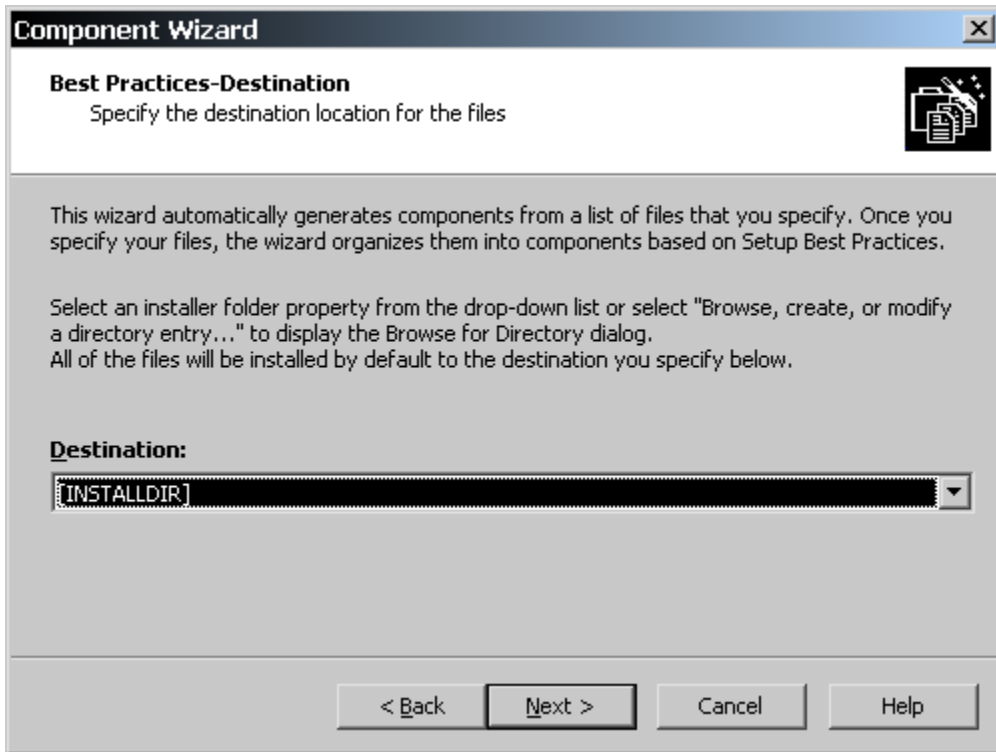


Figure 13-14: *The Best Practices-Destination panel in the Component Wizard.*

You specify a destination by either selecting one of the predefined locations or creating a new location. However, you can change this location later. What you enter here is used to create the component but, because all the components generated by the Component Wizard are available in the project file, you can go to the Setup Design or Components view and define a different destination for any component.

The Files panel is used to select the files and/or folders that contain the files to be placed in the components that are created (Figure 13-15).

The purpose of the files panel is simply to collect all the files that you want the Component Wizard to analyze and then divide them up into appropriate components. You can add individual files or you can add entire folders. If you add a folder, all the files in that folder and all the files and subfolders will be analyzed. You can add folders and then you can select those files you do not want and remove

them. When you select a folder that has subfolders, the name of the subfolder will be appended to the destination that was selected for all the components.

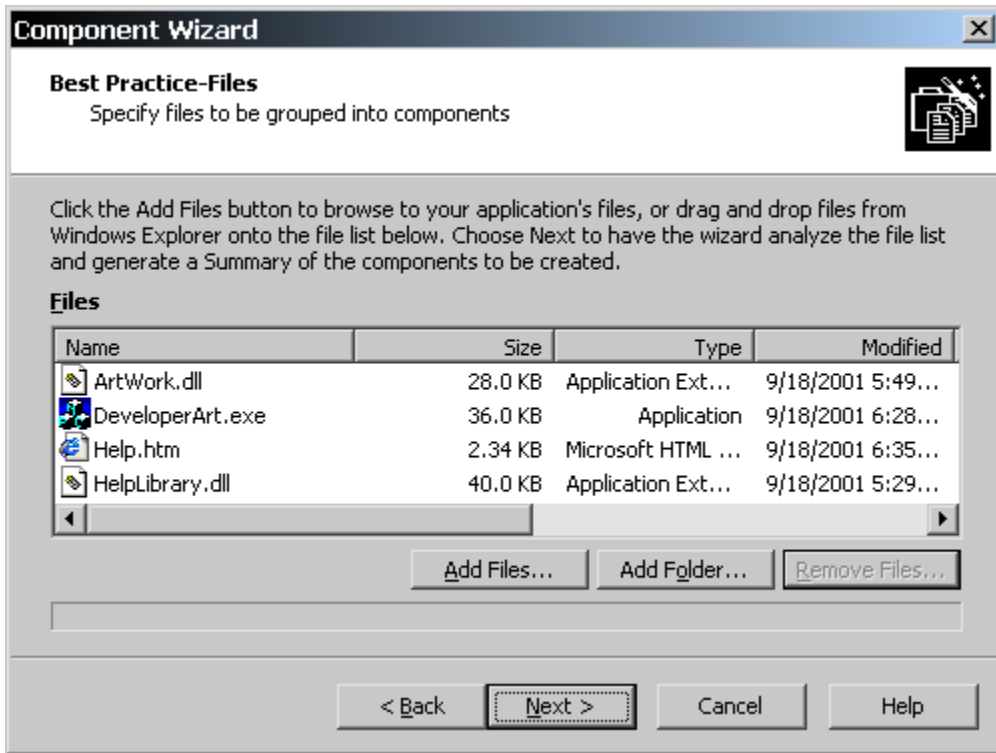


Figure 13-15: *The Best Practices-Files panel in the Component Wizard.*

The analysis process starts as soon as you click the Next button in the Files panel. The first thing that it does is to verify that an available merge module cannot be used to install any of the files. Merge modules are discussed briefly in Chapter 14 but a merge module is a way to encapsulate one or more components so that they can be redistributed.

The locations in which the Component Wizard will look for merge modules are defined in on the File Locations tab of the Options dialog. In the Merge Module Locations edit field is a comma-delimited list of folder locations where merge modules reside on the build system. You can add additional locations to this list by browsing to those locations. By default, three locations are predefined:

- The location where the Microsoft merge modules that ship with InstallShield Developer are installed. This location, by default, is as follows:

```
C:\Program Files\InstallShield\Developer\Modules\i386
```

- The location where the merge modules, created by InstallShield Software Corporation, are installed. This location, by default, is as follows:

```
C:\Program Files\InstallShield\Developer\Objects
```

- The location, on the build machine, where the custom merge modules created by the setup developer are stored. This location is typically set to the following:

```
%USERPROFILE%\My Documents\MySetups\MergeModules
```

If merge modules are placed on the build system in locations that are not specified in the Options dialog, the Component Wizard does not know anything about them. The Component Wizard does not create a component for any file that should be incorporated into the project using an existing merge module.

Once the Component Wizard has ascertained what files need to come from merge modules, it groups the remaining files according to extension and places them in separate components according to the Best Practices rules. The files that do not have the extensions defined in the rules are lumped into one component. The components created for individual files are given a name the same as the name of the file including the extension.

Part of the analysis of the files that the Component Wizard performs is to determine which files are COM servers. Those files that are COM servers have the COM registration information extracted and placed in the proper database tables. For COM servers that are executable files, the Component Wizard looks for the OLESelfRegister string in the version resource. If it is there, COM information is extracted.

When the analysis is complete and the components have been created, the Component Wizard displays the Summary panel (Figure 13-16). This informs you of the results of the component creation and also shows any files that need to be installed using merge modules.

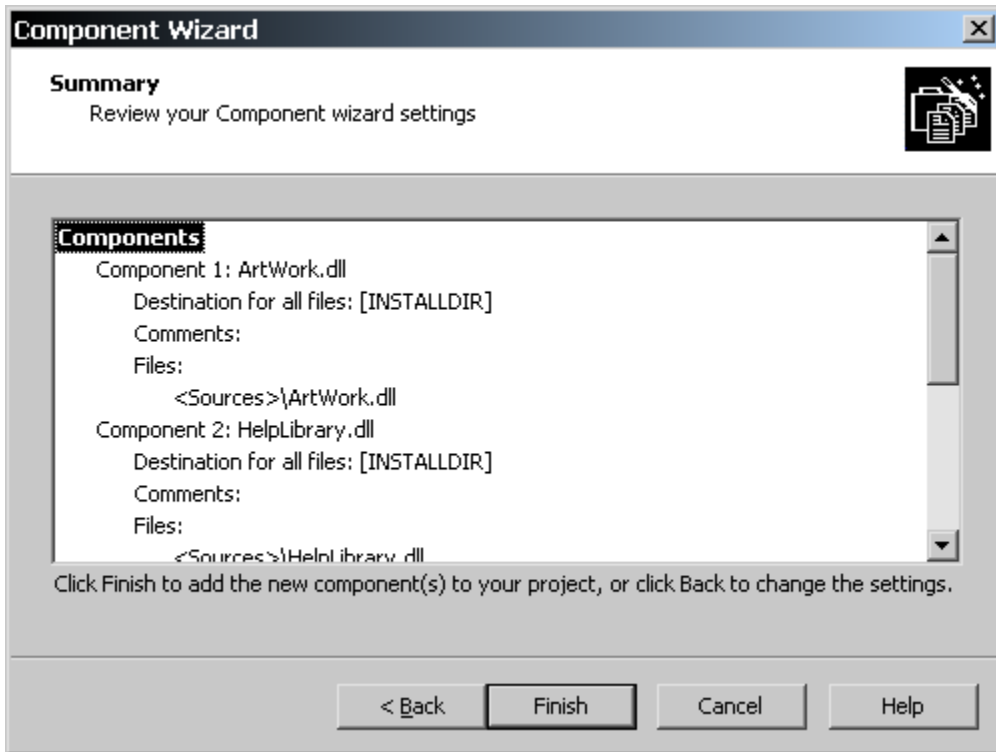


Figure 13-16: *The Summary panel in the Component Wizard.*

Using the Best Practices option in the Component Wizard is a very efficient approach to creating components in an automatic fashion. Since COM is a widely used programming approach, the fact that COM information is extracted as part of the automatic creation of components is a major help in getting a large project up and running.

The next section discusses another way to efficiently create components by scanning applications or by scanning Visual Basic project files.

Scanning

Another handy functionality in InstallShield Developer is the ability to determine what dependencies files might have, so all the needed files can be added to a project. Finding dependent files is implemented in InstallShield Developer through three

different methods of scanning that are available in the Dependencies view under Step 2 in the View List (Figure 13-17). The Static Scanning wizard scans the files already added to a project for any dependencies they may require. This wizard scans all .exe, .dll, .ocx, .sys, .com, .drv, .scr, and .cpl files in a project and allows any detected dependencies to be added to the project. The new files added to a project are added to the same feature as the file that depends upon them, thereby ensuring they are installed when needed.

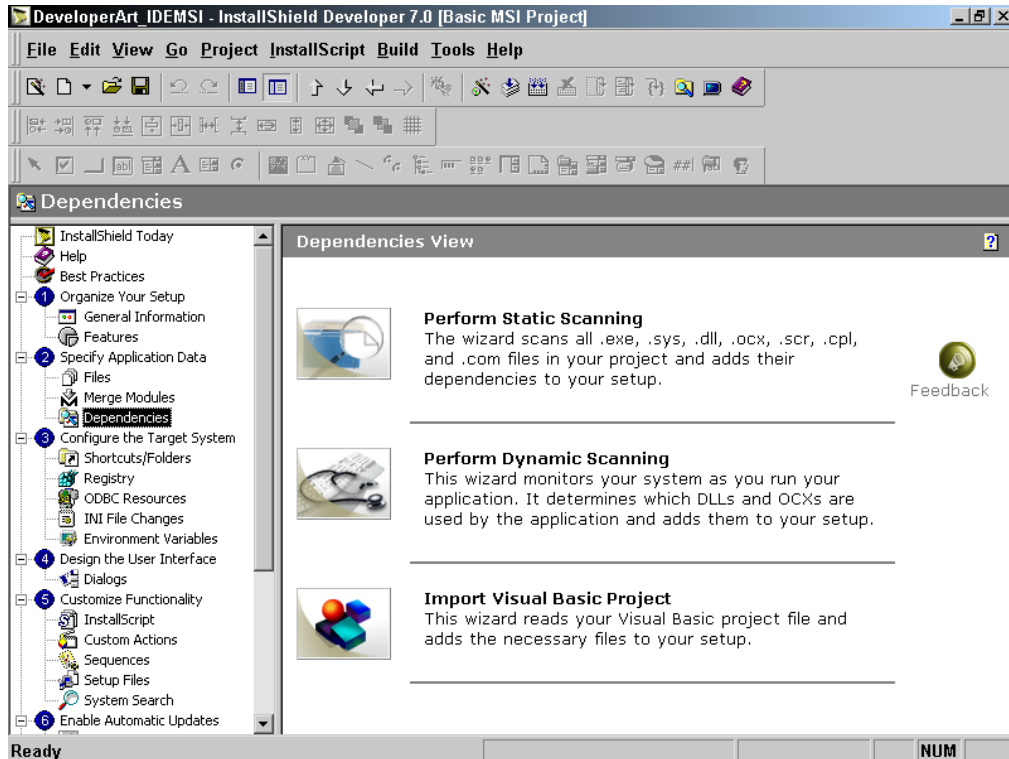


Figure 13-17: *The Dependencies view.*

The Dynamic Scanning wizard is used to add an executable's dependency files to a project. This wizard scans a running executable for all DLL and OCX dependency files and automatically adds them to the project. The wizard can scan for an executable that is already a part of the project, or it can add a new executable to the project prior to starting the scanning process. Dynamic scanning requires you to run the application that you are dynamically scanning. In order to obtain good results, you need to be able to exercise all possible options in this application. This may be easy to

do when the application is not too large, but might present problems for a complex application.

The Visual Basic Wizard is used to import Visual Basic projects into a setup project. The wizard scans the setup project to determine any file dependencies and displays the results of the scan, showing the files that can be added to the setup. The option is then provided for adding the Visual Basic project and its dependencies to the setup project. To scan and import a Visual Basic project, Visual Basic 6.0 must be installed on the build machine. You can launch the Visual Basic Wizard from the Project drop-down menu, in addition to launching it from the Dependencies view. You can also use a Visual Basic project to create a new setup project in addition to just adding components to an existing project.

The wizards used to perform the three types of scanning operations are very similar in their look and feel. Therefore you will experiment with only the Visual Basic Wizard. Before you do this, we need to discuss how to control the file filtering during a scanning operation.

Filtering Files in Dependency Scanners

Functionality common to all three dependency scanner wizards is the option to filter the files that are added to a project. This can be helpful to prevent certain files from being added to projects and then having to remove them. Two initialization files that are named `userscan.ini` and `iswiscan.ini` are used to manage file filtering. These two files are found in the following location:

```
C:\Program Files\InstallShield\Developer\Support
```

The `iswiscan.ini` file contains a list of common system files that can be found on all machines running the Windows operating system. These files are always filtered during a scanning operation. The top part of this file is shown in Figure 13-18.

```
; iswiscan.ini
;
; Files to filter out of static and dynamic dependency scan

[Filter]
6to4svc.dll=1
aaaamon.dll=1
```

Figure 13-18: *The format of the `Isimscan.ini` file*

```

access.cpl=1
Accessibility.dll=1
acctres.dll=1
acledit.dll=1
aclui.dll=1
acsetupc.dll=1
acsmib.dll=1
acssnap.dll=1
activeds.dll=1
activeds.tlb=1
actxprxy.dll=1
admin.dll=1

```

Figure 13-18: *Continued*

This file should never be modified. If you need to override one or more of the files listed in this file, you should use the userscan.ini file. The userscan.ini file contains two sections (Figure 13-19).

```

; userscan.ini
;
; Files to filter/not filter out of static and dynamic scan

; To filter additional files, add one line for each file under
; the [Filter] section.
; For example: (Note: the following line is a sample. When you
; add a line, the semi-colon should not be included)
; MFC42.DLL=1
[Filter]

; To disable the filtering of a specific file listed in iswiscan.ini,
; add a line under [Do Not Filter] section.
; For example: (Note: the following line is a sample. When you add a
; line, the semi-colon should not be included)
; WININET.DLL=1
[Do Not Filter]

```

Figure 13-19: *The format of the Userscan.ini file.*

The userscan.ini file is where you can define your own custom list of files to be filtered. The files that you want to have filtered whenever you perform a scanning operation are placed under the [Filter] section of this file. To override the filtering that is controlled by the iswiscan.ini file, you need to enter the names of the common system files that you do not want filtered under the [Do Not Filter] section. This approach works for all but a small subset of the files listed in

iswiscan.ini. The following list of files will always be filtered regardless of entering them under the [Do Not Filter] section of the userscan.ini file.

- Kernel32.dll
- Ntdll.dll
- User32.dll
- GDI32.dll
- Advapi32.dll
- Shell32.dll
- Ole32.dll

Scanning a Visual Basic Project

This example uses the DeveloperArt_IDEMSI project. The first thing that you need to do is make sure that the temporary feature and component that were added to this project when you were experimenting with dynamic file linking are deleted. On the included CD-ROM there is a small Visual Basic application that you can use to experiment with the Visual Basic Wizard. This Visual Basic application is named GUID Generator and can be used to create GUIDs and also to turn a normal GUID into a packed GUID, as defined at the beginning of this chapter.

With the DeveloperArt_IDEMSI project open, launch the Visual Basic Wizard. Click Next to move past the Welcome panel. The Specify Visual Basic Project File panel appears (Figure 13-20). This dialog allows you to browse to either a .vbp file or a vbg file. You should browse and find the GUIDGenerator.vbp file using the Visual Basic Project edit field. Below this edit field are three options. Normally you will want to select to rebuild the project before the wizard does the scanning to make sure the latest evaluation is performed of the setup project files. For this example, this option does not need to be selected because there is no possibility that any of the Developer Art files are duplicated in the GUIDGenerator application.

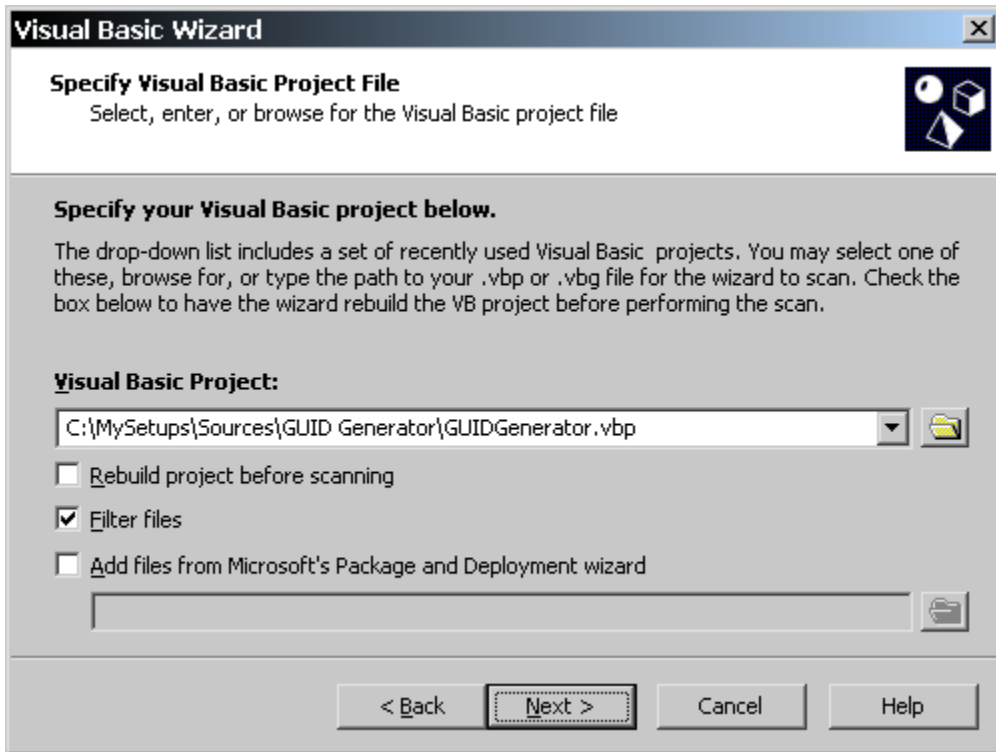


Figure 13-20: *The Specify Visual Basic Project panel.*

The Filter files option is selected by default and this causes the wizard to look in the userscan.ini file to see if there are any entries in this file plus it uses the iswiscan.ini file to filter the common system files.

The last option allows you to identify a dependency file created by the Visual Basic Package and Deployment wizard. Creating a dependency file is one of the options that can be selected when using the Visual Basic Package and Deployment wizard. A dependency file is a text file that has an initialization file structure and part of the information in this file is a list of the dependent files.

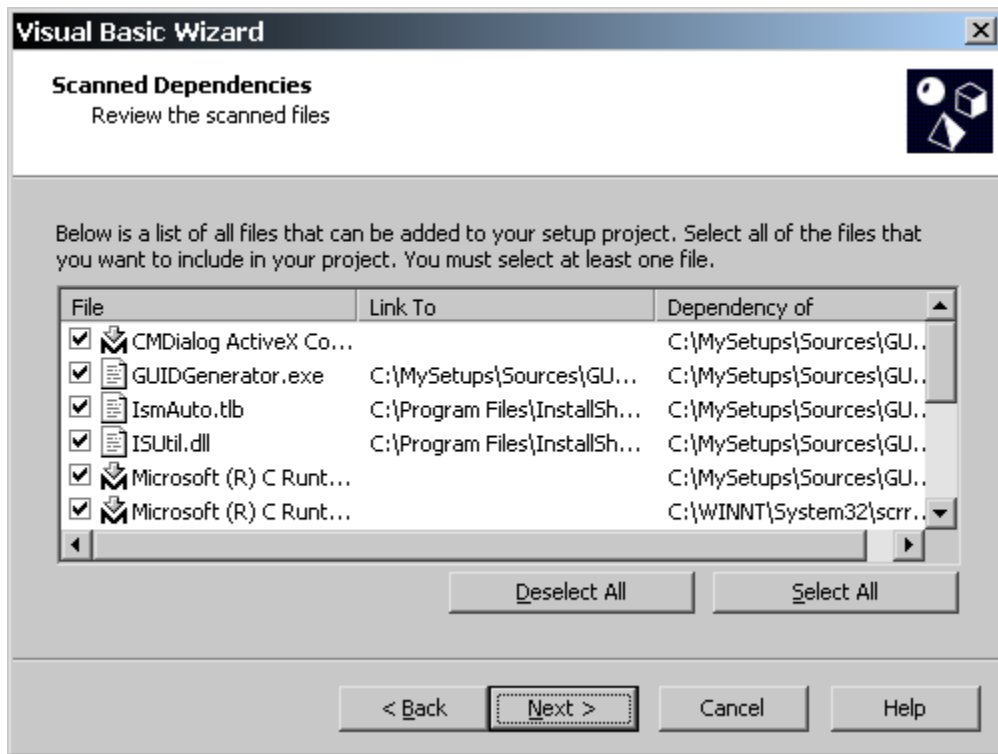


Figure 13-21: *The Scanned Dependencies panel in the Visual Basic Wizard.*

Click Next to begin the scanning process. When the scanning process ends, the Scanned Dependencies dialog is displayed (Figure 13-21). Click the Select All button to add all of the dependencies to the project.

Now click Next to move to the Scan Results dialog (Figure 13-22). This panel shows the scan results and what will be added to the project when you click the Next button. This panel gives you the option of returning to the previous panel and changing the selection that was made there.

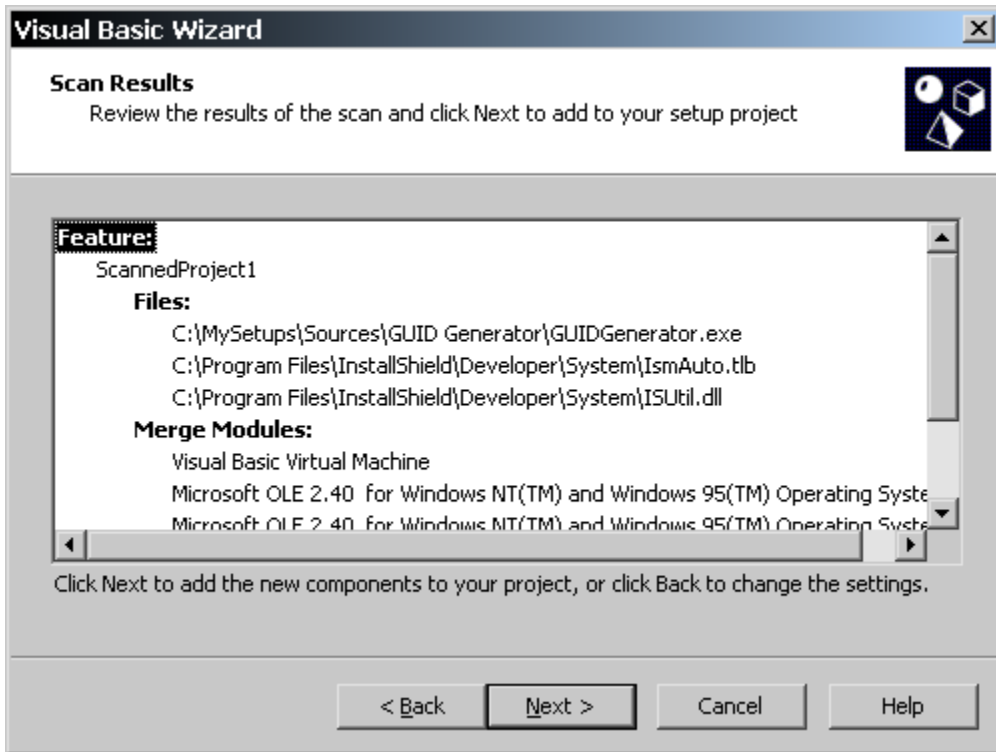


Figure 13-22: *The Scan Results panel in the Visual Basic Wizard.*

Click the Next button to add the dependencies to the project. The wizard displays the Finish panel after the operation is complete.

You can go to the Setup Design view to see that the scanning process has added a new feature called ScannedProject1 and under this new feature are three components for the dependencies that did not already exist in merge modules. The expanded tree under the new feature is shown in Figure 13-23.

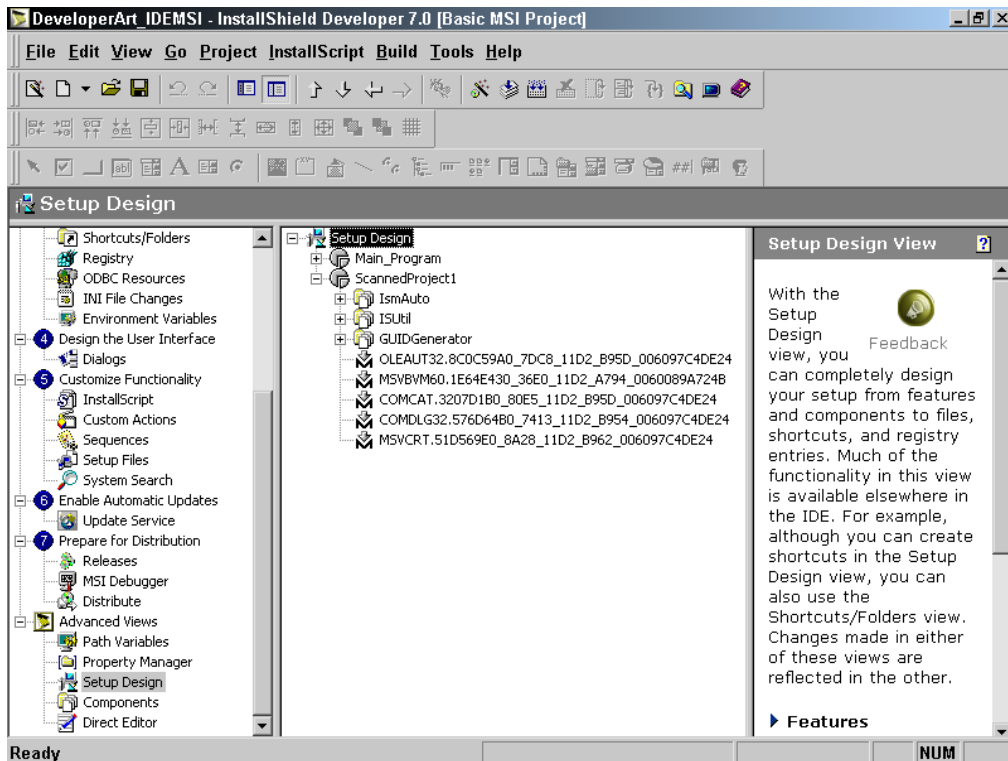


Figure 13-23: *The new feature added by scanning the Visual Basic project.*

There are also five merge modules that are providing components to this feature. All five of these merge modules come from Microsoft and are shipped with InstallShield Developer.

The two components named ISUtil and IsmAuto are components that install files that were originally installed by InstallShield Developer. Since both of these components are COM components, this presents an interesting problem. If you leave these components in the Developer Art installation, it will disable InstallShield Developer when you install it and then uninstall the Developer Art application. This happens because the installation of Developer Art installs these same files to a different location using a different component code than was used in the installation package for InstallShield Developer. Because they are COM components, the entries in the registry would overwrite the entries created when InstallShield Developer was installed.

You could make these components permanent but that would just clutter the target machine. The best solution is to remove these two components from the project. This means that the Developer Art application works only as long as InstallShield Developer is installed on the same machine. You could, of course, create a similar functionality as provided by the automation interface exposed by InstallShield Developer but that is too much work for an example like this.

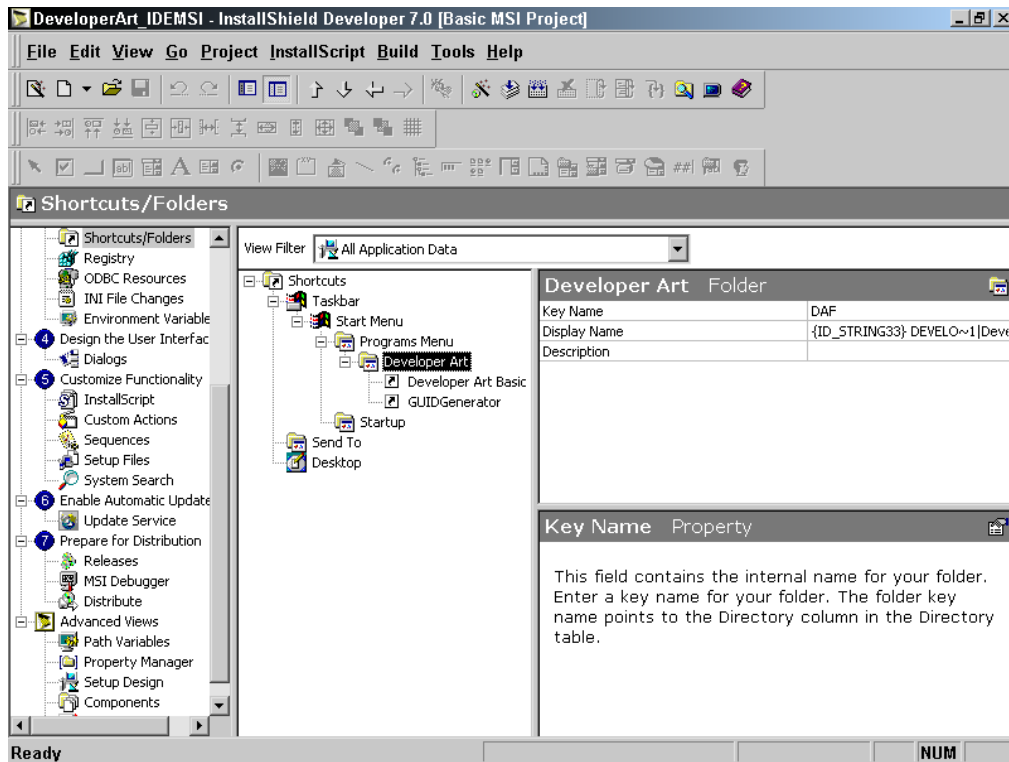


Figure 13-24: *Creating two shortcuts inside a folder for the Developer Art application.*

Next, you should rename the ScannedProject1 feature to something like GUID_Generator. You may want to also add a better display name for the new feature. You should also add a shortcut to the executable in the GUIDGenerator component and then place both shortcuts in the project in a folder that will be created at installation time. This would look like what is shown in Figure 13-24.

The same operations that you just performed in the Basic MSI project can be performed in the same fashion in the Standard project for the Developer Art application. Both projects can be found on the included CD-ROM.

To finish this chapter we will discuss a few items of importance to the creation of components

Special Considerations

There are a number of topics related to the subject of creating and using components. This section discusses these related topics so that you are aware of their existence and their relevance to the installations you create. The first of these related issues is how to maintain a proper reference count of shared components when these components are installed both by applications using the Windows Installer and applications using installation technologies that predate the Windows Installer.

Interfacing with Legacy Applications

At the beginning of this chapter, we discussed how the Windows Installer keeps a count of how many applications have installed a particular component. Prior to the Windows Installer, counting shared files was done differently. Reference counting was done on a file basis and a count of the number of applications installing the same file to the same location was kept under a registry key named SharedDLLs. The full path of this key is as follows:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\  
SharedDLLs
```

Figure 13-25 shows these values that are written under the SharedDLLs key. The value names that are written under the SharedDLLs key are the absolute paths of files that are potentially shared between more than one application. The value data for each file that is registered is a numerical value indicating the number of applications that have installed that same file. In Figure 13-25, you can see that the file MSADDNDR.DLL has been installed by three different applications and thus has a reference count of 3.

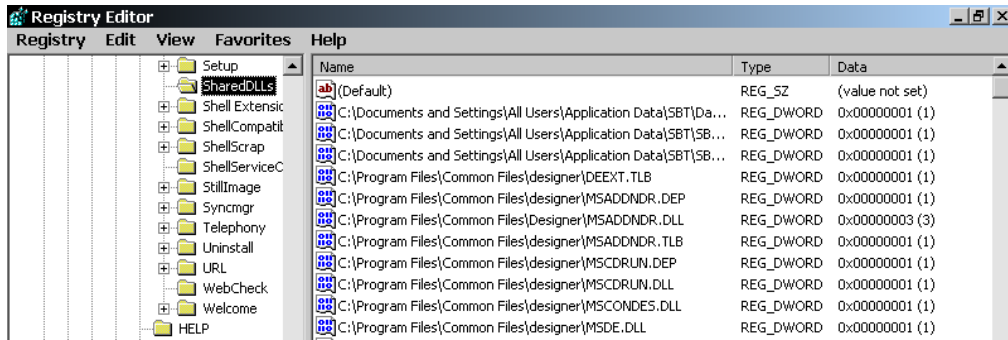


Figure 13-25: Example of the values that get written under the SharedDLLs key.

The mechanism that all install programs were supposed to use would ensure that, during an installation, shared files would get an entry under the SharedDLLs key. If that file were already identified, then the only action would be to increment the reference count for the file. During an uninstallation, the reference count on a shared file that is being removed would be decremented and only if the reference count were 0 would the file be deleted from the system.

The Windows Installer will increment the reference count under the SharedDLLs key when it installs a file that is already listed under the SharedDLLs key. When the Windows Installer is about to remove a component from the system, it checks to see if the file that is being removed is listed under the SharedDLLs key. If it is, the reference count is decremented and only if the reference count is 0 will the component be removed from the system. This ensures that shared files installed by legacy applications versus those same files that are installed by the Windows Installer maintain a link relative to the reference counting mechanism.

The problem is that, by default, the Windows Installer will not create an entry under the SharedDLLs key if it does not exist at the time the installation is performed. The exception to this is that if a component is installed to the System32 folder on Windows NT, Windows 2000, or Windows XP, an entry is made under the SharedDLLs key if it does not exist before the installation is run. For any component not installed to the System32 folder, it is necessary to set a particular property so that an entry is created under the SharedDLLs key if it does not already exist. The name of this component property in InstallShield Developer is Shared. Because of the importance of this property, InstallShield Developer sets the value of this property to Yes for any new component that is created.

There does not appear to be any downside to always having the Shared property set to Yes and there are two scenarios where it is critical that it is set to Yes. The first scenario is where you have a Windows Installer-based application and you have an application installed using an older technology. Both of these installations install the same file to the same location. The following steps show why having the Shared property set to Yes is critical.

1. An application is installed using the Windows Installer and the Shared property is set to No. There is no entry for the file under the SharedDLLs key so one is not created and only the reference counting mechanism for Windows Installer components is in force.
2. Another application is installed using a different technology (legacy application) and, being a good installation, creates entries under the SharedDLLs key for the shared files that make up the application. These shared files all have a reference count of 1. The legacy application installation knows nothing about the reference count created by the Windows Installer based application.
3. The legacy application is now uninstalled and, seeing that the reference count for the shared files is decremented to 0, all the shared files are removed from the system.
4. An attempt is made to run the application that was installed using the Windows Installer and it fails because some of the files that it needs have been removed from the system when the legacy application was uninstalled.

The second scenario where setting the Shared property to Yes is where you have different components (different component codes) that install the same file to the same location. This type of scenario is possible in a large corporation that is making an effort to move to Windows 2000 and is repackaging all their applications to use the Windows Installer. In a large corporation, development activities might be underway all over the country and possibly all over the world. Because of this, it would be easy to create two different components that install the same file to the same location. With the reference counting mechanism used by the Windows Installer, each component would get a reference count of 1 when it is installed. If these components were installed on the same system, then the removal of one of these components would remove the file. Since this file was installed by two different

components, then the component that is left on the system is crippled and the application that depends on this file is also crippled.

The next two sections talk about some of the changes that have been made in the new operating systems to solve the problem of *DLL Hell*.

Windows File Protection

DLL Hell is a term used by Windows developers to refer to the problems that arise when different versions of the same DLL are required by two different applications. One manifestation of DLL Hell is where applications install the version that they need of certain system files without regard to what this might do to the stability of the operating system or to other applications that might be using a different version of these same system files.

With the advent of Windows 2000, a new operating system feature was introduced and is continued as part of Windows ME and Windows XP. This new feature involves a protection scheme for preventing the unauthorized replacement of files that are considered critical to the proper operation of the system. This new functionality is called Windows File Protection (WFP) and it involves keeping a list of files that only Microsoft-approved methods are able to update. The term System File Protection (SFP) means the same thing as WFP, but it is what this protection mechanism was called when it was first conceived.

Windows File Protection uses two mechanisms to prevent the unauthorized replacement of system files. The first mechanism runs as a background process and it monitors any changes that are made in folders where protected files are installed. When the background process receives a notification about a change in a protected folder, it first finds which file was changed. If the file that was changed is on the protected file list, it compares the file signature of the file in the folder with the signature listed in a catalog file. If the signatures do not match, the background process replaces the file with the correct one. In most cases, this replacement is performed out of a cache of files that is kept in the following location:

```
C:\WINNT\system32\dllcache
```

If the modified file is not available from the `dllcache` location, the background process requests access to the original installation media. This can either be a CD-

ROM or a network location. If it is a network location that is available then the replacement happens without any notification to the user.

Each version of the operating systems that have implemented Windows File Protection has a different number of files on the protection list. The included CD-ROM contains a utility that will create a text file showing the details of the file protection list for Windows 2000. This utility may work on Windows ME or Windows XP, but it has not been tested on these operating systems. If you double-click on WFPList.exe, a text file is created that organizes the list of protected files for the current system as shown in Figure 13-26.

```

You are running on Windows 2000 Professional
The version number is 5.0 and the build number is 2195
The service pack level is 1.0

There are a total of 61 directories with a total of 2452
protected files.

The following is a list of the 19 protected file extensions

.axe .axl .axs .axv .cat .cpl .cpx .dll .drv .exe .fon
.icm .inf .ocx .rsp .sys .tlb .tsk .ttf

The following is a list of the protected files relative to the
directories in which they are installed.

c:\program files\common files\microsoft shared\dao\
dao360.dll

c:\program files\common files\microsoft shared\msinfo\
ieinfo5.ocx
msinfo32.dll
msinfo32.exe

c:\program files\common files\microsoft shared\speechengines\tts\
mstssyn.dll
wtss22.dll

...
...
...

```

Figure 13-26: *Partial output from the WFPList.exe utility.*

The output shown in Figure 13-26 is only a very small part of the file produced by the WFPList.exe utility. On Windows 2000 Professional, there are 2452 protected files installed into 61 different folders. The protected files have 19 different file extensions.

The subject of Windows File Protection is much bigger than what has been discussed here. This discussion is only meant to introduce the subject and to get you interested in learning more by going to the Platform SDK documentation.

We now move on to introduce another new feature of the latest operating systems.

DLL Redirection

In the early days of the PC, resources such as hard drive space and RAM were fairly expensive. The design of Windows revolved around the concept of being able to share code between applications in the form of dynamic link libraries. Up until the advent of Windows 98 Second Edition, a dynamic link library would be loaded into global memory once and all applications that used the services of that DLL would have the address range of the DLL mapped to its address space. In this fashion the demand on memory resources was minimized.

This functionality, however, has some undesirable side effects in that when an application loaded a shared DLL all other applications that used that same DLL had to use the version that was in memory. This might be the version that the application needed or it might not. Sometimes having to use another application's version of a shared DLL caused the application to fail. With Windows 98 SE and later operating systems, this requirement to share the code that is in global memory has changed.

Now it is possible with these newer operating systems to have more than one version of the same DLL in global memory at the same time. With memory as inexpensive as it is, this is feasible without degrading machine performance. For all operating systems starting with Windows 98 SE, you can privatize your DLLs and have your own version loaded when your application is launched instead of having to use the version of the DLL that is in the shared location on the machine.

What we are talking about here is forcing the operating system to load a private copy of a DLL instead of loading the version that is in the shared location. This is called DLL redirection or in the Windows Installer world this is called component isolation. The two terms mean the same thing.

When you install an application that has DLLs that are shared with other applications, you still need to copy these DLLs to the shared location on the target machine. You also want to make a copy of these shared DLLs in the same location as the executable or other DLLs that are clients of the shared DLLs. Now you have to force the operating system to load the private copy of the DLL instead of the copy that is in the shared location. This is accomplished by including a zero-byte file that has the same name as the client of the DLL with the addition of a .local extension. For example, if the client executable has the name MyApp.exe, then the zero-byte file will have the name MyApp.exe.local. The presence of this file forces the operating system to load the local copy of the DLL instead of the one that is in the shared location.

The presence of the .local file even overrides any absolute paths that are used for loading the DLL. This is applicable to both Win32 DLLs where the `LoadLibrary` function may have an absolute path as well as the absolute path written to the registry when a COM server is registered. The Windows Installer makes it easy to create the .local file.

To have the Windows Installer make a copy of the shared DLL in the same location as the client and to also create the .local file, you need to make entries in the `IsolatedComponent` table. The `IsolateComponents` action reads this table and sets up a duplicate file operation in order to copy the shared DLL to the private location. The same action also creates the .local file in the private location.

You can author the `IsolatedComponent` table in the Direct Editor view. This table has only two columns where you place the name of the shared DLL component in the first column and the name of the client component in the second column. For the component that is installing the shared DLL that is going to be privatized, it is important to set the `Shared` property to `Yes`. This ensures that a privatized DLL is not uninstalled when it should not be.

Transitive Components

A transitive component is one that has the condition on it reevaluated during a maintenance installation. Unless the `Reevaluate Condition` property is set to `Yes`, the condition on a component is evaluated only when it is first installed. If the changes to the environment change the results of the condition statement, the component will still be enabled because the condition is not reevaluated for every environmental change. Setting the `Reevaluate Condition` property to `Yes` forces the Windows

Installer to reevaluate the condition on the component during every maintenance operation.

A typical use for transitive components is to prepare a product to gracefully reinstall during an upgrade from Windows 98 to Windows 2000. The setup developer specifies those components that need to be swapped out during a system upgrade by setting the Reevaluate Condition property to Yes. When the end user later upgrades the system from Windows 98 to Windows 2000, the product has to be reinstalled. Upon this reinstall, the installer removes the Windows 98 components and installs the Windows 2000 components. This prevents requiring the end user to reinstall the entire product.

Qualified Components

A qualified component is a method of creating an array of components. Qualified components are primarily used to group components with parallel functionality into categories. Qualified components are created in the same way as ordinary components. Every component must have a unique component ID GUID and component identifier specified in the Component table. In addition, qualified components are associated with a category GUID and a text-string qualifier in the PublishComponent table. The category GUID and the qualifier, which just points to the ordinary component in the Component table, reference qualified components.

The Orca database-editing tool provides an example of how qualified components are used. In Orca there are three components that perform different levels of database validation. These components have a parallel functionality and are grouped together under a single category GUID in the PublishComponent table. Each row in this table assigns a different qualifier string that acts like an index into the array of components. There is also a different text string associated with each of the components that are part of the array.

The category GUID, qualifier, and text string are entered into the registry during installation for each component that gets installed. Applications can then retrieve this information from the registry and use the text string to populate controls in the applications user interface. In Orca the text strings for the three components that are part of the array are used to populate the drop-down menu in the dialog that is launched when you select the Tools\Validate option.

You should open the Orca.msi file using Orca and first look at the PublishComponent table. Then launch the Validation Output dialog from the Tools\Validate menu item. You will see that Orca queries the registry to see which components in the array of components were installed and then offers the available options to the user.

Companion Files

Many files that are installed do not contain a version resource. In this case, the file versioning rules of the Windows Installer use a comparison between the created and modified dates to decide if a file already on the system should be replaced.

One way to provide a different functionality for a non-versioned file is to tie its installation to the version of a parent file that does have a version. When you do this, the non-versioned file becomes a companion of the versioned file.

The installation state of a companion file depends not on its own file versioning information, but on the versioning of its companion parent. To specify a companion file, the primary key of the companion parent in the File table must be authored into the Version column of the record for the companion. In other words, you set the Version column in the File table to be equal to the primary key in the file table of the file that does have a version.

To define a companion file, go to the file you want to be a companion in the setup project, right-click on the file name, and select the Properties option. As an example, Figure 13-27 shows how to set the file FirstSound.wav as the companion to the Sounds.exe file in the InstallSoundsServices component. This is a component that you will create in Chapter 14 when we discuss NT services.

For the FirstSound.wav file, you need to first deselect the “Use system attributes” option and then enter the name of the primary key being used in the File table for the Sounds.exe file. The key is shown in the Files view for the component. Primary keys are case sensitive, so you need to enter the name of the key exactly as shown in the Files view.

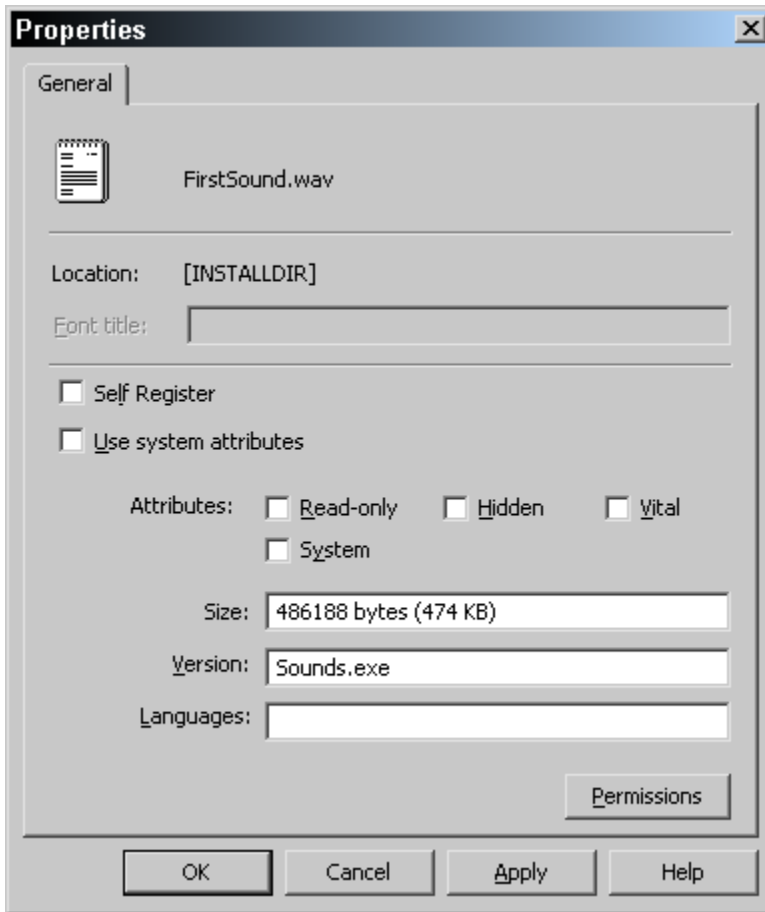


Figure 13-27: *Defining a companion file using the file Properties dialog.*

Companion files can be files that have a version but that you do not want to have used during the file copy. Also, a companion file does not have to be in the same component as the parent file. However, you cannot make a file that is the key path for a component the companion of another file.

Conclusion

This chapter provided you with an introduction to components. Understanding components is at the heart of creating installation packages that work well with the Windows Installer technology. This chapter began with a discussion of how the Windows Installer registers a component and how it uses the component code. This was followed by a discussion that presented the rules that should be followed for the creation of components.

Then the tools that are part of the functionality of InstallShield Developer were introduced, starting with dynamic file linking and the scanning mechanisms that are available. Following this, the Component Wizard was introduced showing that there are two modes of operation.

The end of the chapter presented a number of special topics that introduced certain technical concepts that are important to understand in conjunction with the installation of components. Each of the topics presented are worthy of further study because of their importance.

Chapter

14

Creating Special Components

In the last chapter we took a look at the basics of component creation. In this chapter we will discuss the creation of four special types of components. We will look at using the Component Wizard to create components that install COM servers, NT services, and fonts. We will then examine the use of the ODBC Resources view to create components that install ODBC drivers, translators, and data source names. At the end of this chapter we will take a brief look at merge modules and you will create a simple merge module for the ArtWork component.

COM Components

COM servers require a heavy use of the registry. The creation of a COM component revolves around the acquisition of the correct registry keys and values required by a particular COM server for it to function correctly. As an example, you will use the Component Wizard to create another version of the ArtWork component, which installs the COM server ArtWork.dll. You will have to remove the present version of the ArtWork component from the DeveloperArt_IDEMSI project.

Before you do this, however, there are a number of issues related to installing COM servers that we need to discuss. This book covers only *in-process* servers. We begin our discussion of COM issues with a short description of the differences between a normal Win32 DLL and a COM DLL. This discussion is very basic and is meant to provide a sense of what is happening with both types of DLLs. When working with COM, it is important to be able to put it into perspective relative to Win32 DLLs. This discussion will also make it easier to understand the concepts of DLL Redirection (isolated components) and side-by-side component sharing. DLL Redirection is discussed at the end of Chapter 13.

Win32 DLLs vs. COM DLLs

As background for the upcoming discussions concerning the COM component creation, it is appropriate to delve into the differences between a standard (Win32) DLL and a COM DLL. We use the term Win32 to indicate programming that uses the Windows 32-bit API and that is compiled to run only on 32-bit Windows operating systems. Dynamic link libraries have been around since the inception of the Windows operating system. DLLs were created in order to be able to share code between applications and this was important in the days when computer resources such as RAM and hard drive space were expensive. Even though computer resources are no longer as expensive, the modular approach to creating applications is still the way programming is performed.

Exporting Functions From a DLL

In a typical application, the main executable of the application is termed the client. The various DLLs that make up the rest of the application's functionality are termed the servers since they provide functionality that the executable needs. This

functionality is provided in the form of functions that the executable can call. However, for an executable to use a function that has been defined inside of a DLL the DLL needs to export the names of the functions so that the executable can get the address in memory of where the code for the function begins. Functions that are exported from a DLL are listed in a special part of the header to the DLL file. The name of this special part of the DLL's file header is the Export Table.

There is a utility that comes with Microsoft Visual C++ that can be used to display the names of the exported functions in a dynamic link library. This utility is DUMPBIN.EXE and can be found in the Bin folder under where Visual C++ is installed. If you use this utility to display the exported functions from the two DLLs that are part of the Developer Art application, you will see what is shown in Figure 14-1.

```
Microsoft (R) COFF Binary File Dumper Version 6.00.8447
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

Dump of file ArtWork.dll

File Type: DLL

Section contains the following exports for ArtWork.DLL

    0 characteristics
3BA77B35 time date stamp Tue Sep 18 12:49:57 2001
    0.00 version
    1 ordinal base
    4 number of functions
    4 number of names

ordinal hint RVA      name
1         0 0000105B DllCanUnloadNow
2         1 00001067 DllGetClassObject
3         2 00001081 DllRegisterServer
4         3 00001091 DllUnregisterServer
```

Figure 14-1: *The export tables for the ArtWork.dll and HelpLibrary.dll files.*


```

Summary
    1000 .data
    1000 .rdata
    1000 .reloc
    2000 .rsrc
    1000 .text

Microsoft (R) COFF Binary File Dumper Version 6.00.8447
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

Dump of file helplibrary.dll

File Type: DLL

    Section contains the following exports for HelpLibrary.dll

     0 characteristics
3BA77663 time date stamp Tue Sep 18 12:29:23 2001
    0.00 version
     1 ordinal base
     1 number of functions
     1 number of names

ordinal hint RVA      name
    1     0 00001010 _LaunchHelp@4

Summary
    4000 .data
    1000 .rdata
    1000 .reloc
    4000 .text

```

Figure 14-1: *Continued.*

ArtWork.dll is a COM DLL and HelpLibrary.dll is a Win32 DLL. You can see from Figure 14-1 that both of these DLLs export the names of functions that can be called by clients. However, there is a big difference in the purpose of the functions exported from a COM DLL than those exported by a Win32 DLL.

In HelpLibrary.dll, there is only one function that is exported and the exported name of this function is `_LaunchHelp@4`. There are four functions that are exported by

the COM DLL `ArtWork.dll` and none of these functions directly expose the functionality contained in the DLL. The purpose of these exported functions is to enable the services of the COM DLL to be accessed when the client of these services needs to call on them. How the services of both a Win32 DLL and a COM DLL are accessed is discussed in the following two sections. This is important information because it makes it clearer why certain actions are taken during an installation.

Accessing The Functions In a Win32 Dll

There are two methods for accessing the functions that are exported by a Win32 DLL. There is load-time dynamic linking and there is run-time dynamic linking. The type of dynamic linking used by the Developer `Art.exe` application is load-time dynamic linking. We will discuss load-time dynamic linking first, and then we will look at run-time dynamic linking.

When the function in a DLL is called explicitly, this forces the DLL to be loaded into memory when the client of the DLL is loaded into memory. Developer `Art.exe` explicitly calls a function name `LaunchHelp` and, when it is compiled, it links to the import library of the `HelpLibrary.dll`. The name of the explicitly called function is entered in to another special location of the file header for Developer `Art.exe`. The name of this special location is the Import Table and the data in this table can also be viewed by using the `DUMPBIN.EXE` utility. The very first part of the Import Table for the Developer `Art.exe` file is shown in Figure 14-2.

```
Microsoft (R) COFF Binary File Dumper Version 6.00.8447
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

Dump of file DeveloperArt.exe

File Type: EXECUTABLE IMAGE

Section contains the following imports:

  HelpLibrary.dll
    403000 Import Address Table
    403C44 Import Name Table
    0 time date stamp
    0 Index of first forwarder reference
    0 _LaunchHelp@4
```

Figure 14-2: *The first part of the Import Table in Developer `Art.exe`.*

The name of the function in the source code that is exported by `HelpLibrary.dll` is `LaunchHelp`, but the Visual C++ compiler decorated the exported name of this function to be `_LaunchHelp@4`. The compiler performs name decoration of exported functions when the client executable explicitly calls the DLL function, which is the case with `Developer Art.exe` and the function exported by `HelpLibrary.dll`. You can see this decorated function name in both the Export table of `HelpLibrary.dll` shown in Figure 14-1 and in the Import table of `Developer Art.exe` shown in Figure 14-2.

Because `Developer Art.exe` explicitly calls the function exported by `HelpLibrary.dll`, it needs to be able to load this DLL into memory whenever you launch the application. If the `HelpLibrary.dll` is not available when you launch the application, then you will receive an error message and the application will not load. When you launch a program that uses load-time dynamic linking, the operating system looks in the Import table to find the names of the DLLs that are required. After obtaining the names of the required DLLs, the operating system looks in order in the following locations to find these DLLs. The operating system searches the following locations in the sequence that they are presented in the following list.

- The folder from where the application was launched.
- The current directory.
- The Windows system folder.
- The Windows 16-bit system folder.
- The Windows folder
- The folders specified by the `PATH` environment variable.

The last location is important because this is what the `App Paths` key in the registry modifies for each individual application. In Chapter 5, you added a value to the `Application Path` property for the `DeveloperArt` component. The `Paths` value is appended to the `PATH` environment variable when the application is launched.

When an application uses run-time dynamic linking, there is no Import table entry for the DLL that will be accessed. Instead, the client of the DLL loads the DLL into memory using the `LoadLibrary` Windows API. A pointer is then obtained to the

exported functions that need to be accessed by using the `GetProcAddress` Windows API. If the `LoadLibrary` API is passed just the name of the DLL without qualifying its location, the Windows operating system searches for the DLL in the same locations and sequence as listed above. The `GetProcAddress` API uses the name of the exported function as an argument in order to return the address in memory of the function. This means that the DLL needs to be created in such a way that the compiler does not decorate the name of the exported function, so that the name of the exported function is known in advance. Using the `LoadLibrary` API to load a DLL into memory and then getting the address of an exported function using the `GetProcAddress` API is how the Windows Installer accesses a function in a DLL that is used as a custom action.

Accessing The Services In a Com Dll

There are four functions, as shown in Figure 14-1, that are exported from the COM server `ArtWork.dll`. None of these functions exposes the functionality of the COM server, but only allows that functionality to be accessed. This section discusses briefly what happens when a client wants to access the services of a COM server. Then you will learn what has to be accomplished during an installation so that accessing these services is possible when the application is launched. There is one point of terminology that we need to clarify and that is when we talk about the services that are provided by a COM server, we usually talk about calling methods (not functions) and getting or setting properties.

The names of the methods in `ArtWork.dll` that are used to draw the various geometrical shapes are named `CreateCircle`, `CreateRectangle`, and other similarly named methods. These methods are exposed to the client, in this case Developer `Art.exe`, through an *interface*. An interface to a COM server is a table of pointers to the methods that are implemented by the COM server. The job of the client is to get a pointer to this interface so that it can avail itself of the methods that are implemented by the COM server. A pointer to an interface is actually a pointer to a pointer to the table that holds the pointers to the methods implemented by the COM server. To call a method in a COM server, the client essentially gets a pointer to the interface, dereferences that pointer so that it now has a pointer to the table of the exposed methods, and then uses that table pointer to point to the method that it wants to call. The next section discusses how a client finds the location of the COM server and then gets the interface pointer that it needs in order to call the required methods. The process of obtaining the interface pointer to a COM server relies on

being able to access functions in the COM library, which is part of all 32-bit Windows operating systems. The COM library is implemented by OLE32.DLL found in the Windows system folder.

The steps of the process used to create a COM object in memory and then to access a pointer to its interface are shown in the following list:

1. The client initializes the COM library by calling either the `OleInitialize` or the `CoInitialize` functions exposed by OLE32.DLL.
2. The client begins the process of creating the COM object by calling the `CoCreateInstance` function. This function in turn calls the `CoGetClassObject` COM library function.
3. Two of the arguments that are passed to the `CoGetClassObject` function are the GUID that represents the Class ID for the COM server and the type of execution context in which the COM server is to run. When a COM server is implemented as a DLL, the execution context will be called `InProcServer32` and registered as such in the registry. Using this Class ID and the execution context, the `CoGetClassObject` function searches the registry to obtain the name and location of the DLL that implements the COM interface that the client requires. When the DLL is found from the registry, the `CoGetClassObject` function loads the DLL into memory using a function similar to the `LoadLibrary` function discussed previously. When the DLL is loaded into memory, the address of the `DllGetClassObject` function is obtained and a call is made to this function.
4. The `DllGetClassObject` function creates a class factory. A class factory is a COM object that exposes methods through a special interface defined by the COM library. The `DllGetClassObject` function also returns a pointer to the class factory back to the `CoCreateInstance` function. The `CoCreateInstance` function then uses this pointer to call the `CreateInstance` method that is implemented in the class factory interface.

5. The `CreateInstance` method creates an instance of the COM object that implements the methods that the client needs to call. When the `CreateInstance` method is finished creating the COM object, it provides a pointer to the interface of this component object to the client. The class factory COM object is then destroyed because it is not needed anymore.
6. When the end user is finished with the application and closes it, the client calls either the `CoUninitialize` or the `OleUninitialize` functions in the COM library. Part of the process of shutting down will be for these functions to call another function that is exported from the COM DLL. The name of this exported function is `DllCanUnloadNow` and it lets the COM library know if there are any other clients using the services of the COM DLL. If no other client is using it, the COM DLL is unloaded from memory; otherwise, it is left in memory.

The above discussion provides a basic knowledge of how the mechanism used with COM DLLs differs from the mechanism used with Win32 DLLs.

When a COM server is implemented in an executable, the client runs in one process and the COM server runs in a different process. The mechanism for implementing COM is slightly different in this scenario because it requires communication across process boundaries. No exported functions are called because an executable does not export functions like a DLL does. However, a COM server implemented as an executable is still found by the client by calling the same COM library functions. The COM server still sends across the process boundary a pointer to the interface that implements the methods required by the client.

If you go to Figure 14-1, you will see that there are two functions, `DllRegisterServer` and `DllUnregisterServer`, exported from `ArtWork.dll` that we have not discussed yet. The purpose of these two functions is the topic of the next section.

Self-Registration Of a COM Server

The last section explained that, when a COM DLL is to be loaded into memory, the location is found by the COM library function `CoGetClassObject`. This

function looks in the registry using the Class ID and execution context of the COM object to be instantiated to find this location. What the registry looks like for the `ArtWork.dll` is shown in Figure 14-3.

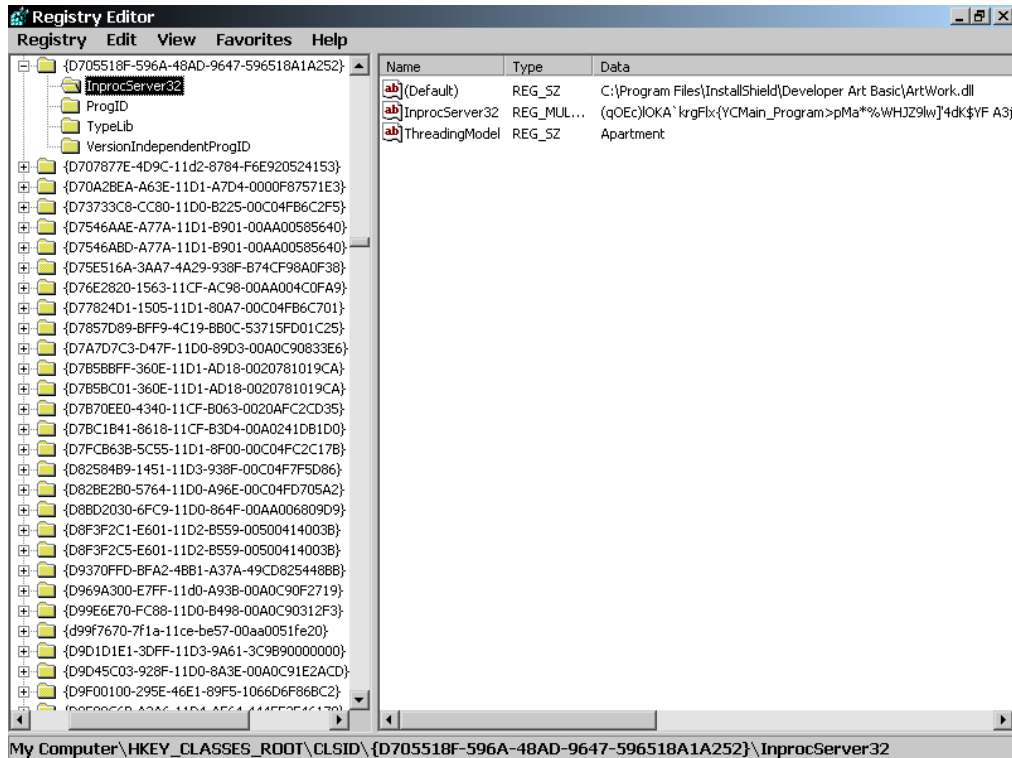


Figure 14-3: *The registration for the `ArtWork.dll` COM server.*

The default value for the `InprocServer32` registry key is the absolute path to `ArtWork.dll` COM server. The `InprocServer32` registry key is a sub-key of a key that is named using the Class ID for the component object that the client needs to instantiate. This is the information in the registry that is used to load this DLL into memory. This information is added to the registry when the DLL is installed. The registry entries for a particular COM server are defined in the `DllRegisterServer` function that is exported by a COM server.

An executable can load the COM DLL into memory using the `LoadLibrary` Windows API and then call the `DllRegisterServer` function. The

`DllRegisterServer` function writes all necessary entries into the registry. When these entries are made to the registry during an installation, it is the installation program that loads the DLL into memory and calls the `DllRegisterServer` function. The term self-registration refers to the fact that a COM DLL or a COM EXE contains the code that makes the registry entries that clients need in order to access the services implemented by the COM server. The `DllUnregisterServer` function is called when an application is being uninstalled in order to remove the registry entries that were created during the installation.

Because an executable does not export any functions, there needs to be another approach to telling it to create the required registry entries. The approach that is used is to pass a command line argument to the executable that tells it not to run in its normal mode, but to create only the necessary registry entries. The switch that is used to tell the executable to create the registry entries is `/regserver` and the switch that is used to tell the executable to remove the registry entries is `/unregserver`.

Until the advent of the Windows Installer technology, self-registration was the only method that could be used to make the necessary COM-related registry entries during an installation. There were often problems with self-registration because, in real-world applications, there were many DLLs that loaded other DLLs, creating a chain of dependency. It then became necessary to register the dependent DLLs in the correct order. With the Windows Installer, the database approach provides a more robust means for registering COM servers.

With the Windows Installer, it is not necessary to use a COM servers' self-registration capabilities during an installation because all the required registry entries are placed in database tables. Using the values in the database tables, the Windows Installer takes these values and writes them to the registry. It is not necessary to load a DLL into memory and then execute the `DllRegisterServer` function. The real work comes when the installation package is built and the COM registration information is extracted from the COM servers in order to populate the database tables. It is now the build process that needs the self-registration functionality and not the installation process. The extraction of COM information by InstallShield Developer is made possible because of the implementation by DLLs of the `DllRegisterServer` and `DllUnregisterServer` functions. For executables, the same thing is true. It is just a matter of capturing the information that the COM servers write to the

registry when the build is made and then placing that information into the database tables.

Just because the Windows Installer has a better approach to registering COM servers does not mean that self-registration is not supported. The Windows Installer database schema contains a table named `SelfReg` and the standard actions `SelfRegModules` and `SelfUnRegModules`. By adding rows to the `SelfReg` table, the two standard actions register the DLLs that are identified in the table during installation and unregister them during uninstallation. There are two problems with the implementation of self-registration by the Windows Installer. The Windows Installer is able to self-register DLLs, but it cannot self-register executables. Also, Windows Installer requires custom actions in order to control the order of DLL registration.

InstallShield Developer has implemented a more robust mechanism for self-registration. This mechanism allows for the self-registration of executables and allows you to control the order of self-registration of COM servers. When you right-click on a file in a component and display the Properties dialog, the dialog contains a Self Register option. When you select the Self Register option, the name of the file is placed in a custom table named `ISSelfReg`. This table can be accessed using the Direct Editor view and the order of registration can be controlled through entries in the Order column. There is also a `CmdLine` column in the `ISSelfReg` table where you can add special command line options for self-registering executables. By default, an executable that appears in the `ISSelfReg` table is passed the `/regserver` switch during installation and the `/unregserver` switch during uninstallation.

There are a number of reasons why you should not self-register your COM servers and these are discussed in the next section. The only reason that you may need to self-register a COM server is if there are registry entries that are made as part of self-registration that are not related to COM.

Why Self-Registration is Not Recommended

We have just spent some time discussing the concept of self-registration and how COM servers implement it. Now we need to understand why Microsoft does not recommend using self-registration. The best way to characterize the difference between self-registration and the approach used by the Windows Installer is that self-

registration is a black box and placing the registry entries into the database tables identifies specifically what registry entries are to be made. Self-registration is a black box in that all the Windows Installer knows is to call the `DllRegisterServer` function during an installation and to call the `DllUnregisterServer` function during an uninstallation. It does not know anything about what registry entries are created or removed.

There are a number of reasons that the black box approach to registering COM servers is undesirable. The following are the most important reasons.

- One of the important features of the Windows Installer is to be able to treat an installation as a transaction. With the transaction approach it is possible to rollback any changes if an installation is aborted for any reason before it is finished. A safe rollback becomes problematic when self-registration comes into the picture because the Windows Installer does not know if any of the registry entries created during the self-registration of a component are used by any other components when it is time to rollback any changes. All the Windows Installer can do during the rollback is call the `DllUnregisterServer` function. This has the potential to disable components installed by other features or applications.
- Advertisement is another major feature of the Windows Installer technology where an application is not installed until it is needed. Installation of an advertised application can be initiated by the activation of a COM server in an advertised component. This is because when an application is advertised, the values in the `Class`, `TypeLib`, and `ProgId` tables are written to the registry. If the COM server uses self-registration, these registry entries are not made when advertising an application and thus the advertisement functionality is broken with respect to the self-registered component.
- Chapter 10 explained that it is possible to define registry entries that can be made under either the `HKEY_LOCAL_MACHINE` root registry key or the `HKEY_CURRENT_USER` registry key depending on whether the end user selects to perform a per-machine or a per-user installation. This is not supported when a component uses self-registration because the `DllRegisterServer` function does not support the concept of having a per-user registry key for COM class registration.

- If multiple users are using an application on the same computer that has self-registered modules, each user must install the application the first time they run it. This is because the Windows Installer cannot determine that the proper HKEY_CURRENT_USER registry keys exist.
- If an application is being installed from an administrative image on a network drive, the `DllRegisterServer` function can be denied access to network resources such as type libraries if a component is both specified as run-from-source and is listed in the `SelfReg` table. In this event, the component can fail to register properly and thus the application is disabled in whole or in part.
- As mentioned in the previous section, sometimes there is a chain of dependencies between DLLs that need to be registered. If registration does not take place in the proper order, it is not possible to register some or all of the DLLs. This is because it is necessary to load a DLL into memory before it the to call to the `DllRegisterServer` function can be made. When the database tables are used, the problem of dependencies is avoided because it is not necessary to load any of the DLLs into memory. The required registry entries are written to the registry out of the database tables.

Sharing COM Components

Chapter 3 stated that components are a shareable entity. This is true, except in cases where you try to share a COM component between two different features of the same application. There is a problem with the schema of the `Class` table that prevents you from sharing COM components between features. For this discussion, it is assumed that you do not want to self-register the COM component because of the reasons stated in the previous section.

Because the source of the problem is with the `Class` table, we need to look at its schema. Repeated here is the schema of the registry tables that was first shown in Chapter 3 (Figure 14-4).

The schema of the `Class` table is shown in the lower left. From this diagram, you can see that there are only four columns in the `Class` table that are not allowed to be NULL. These are the `CLSID`, `Context`, `Component_` and the `Feature_` columns. For

any COM component, the first three of these columns form the primary key for the table.

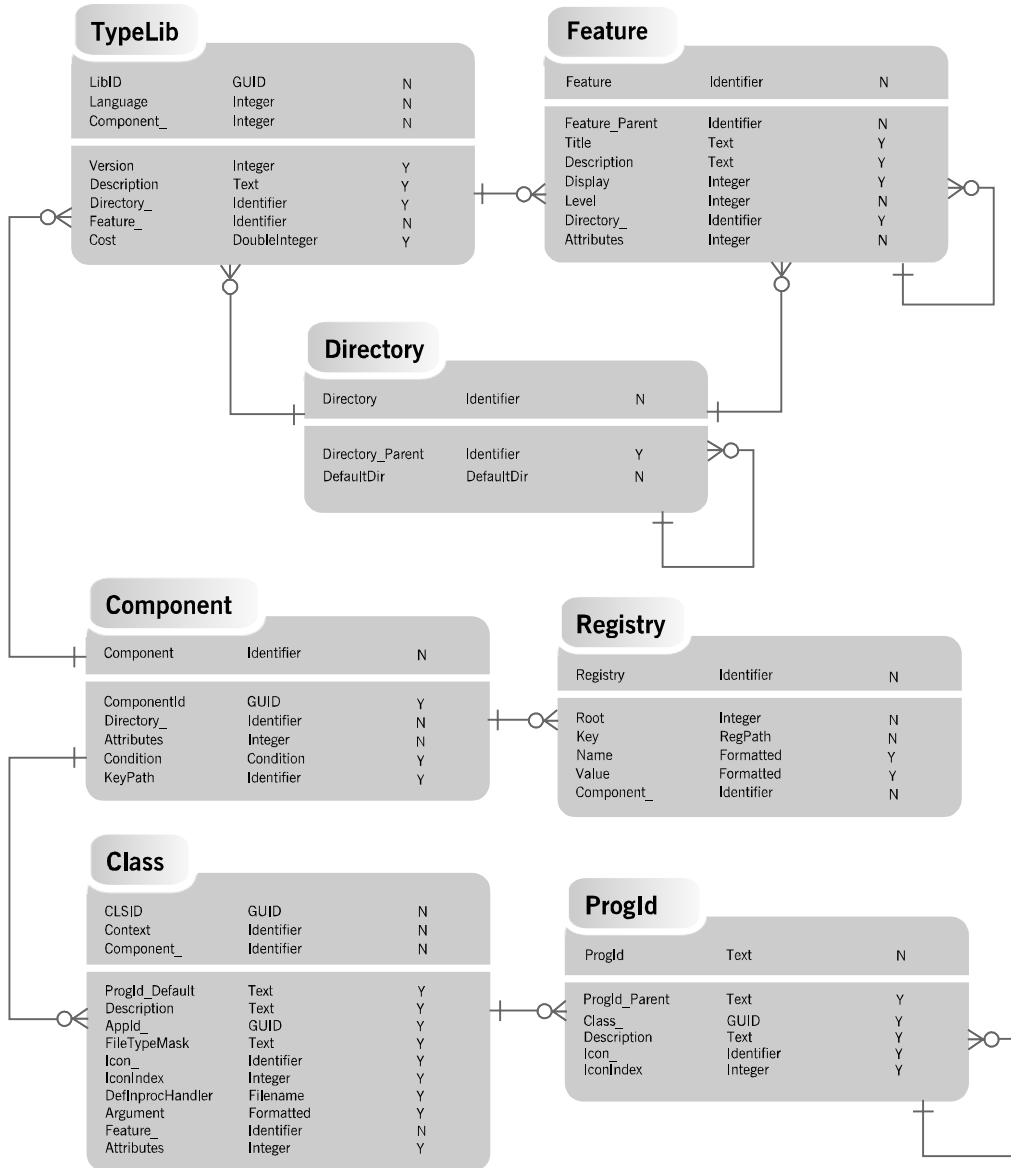


Figure 14-4: The schema of the registry related tables.

If you try to assign a COM component to two different features, you would need to enter the same values for the first three columns. Only the entry for the Feature_ column would be different. The problem arises because, in a relational database, you are not allowed to have duplicate primary keys and this occurs because the name of the feature is not part of the primary key.

The solution to this problem is to create your application structure so that there is one top-level feature and that feature is set as required. In other words, you have the top-level feature where the end user cannot deselect it in the custom setup dialog. You then assign your COM component to that feature to ensure that it is always installed. The COM component's functionality is available to any other feature that needs it. The only time that this COM component will be removed from the system is when the entire application is uninstalled.

The next section looks at how the Component Wizard creates a COM component.

Recreating the ArtWork Component

In this section, you will use the DeveloperArt_IDEMSI project to see how to use the Component Wizard to create an individual COM component. To start off open the DeveloperArt_IDEMSI project and delete the ArtWork component. You delete a component from a project by right clicking on the component in the Setup Design view and selecting the Delete from project option.

To launch the Component Wizard go to the Setup Design view, right-click on the MainProgram feature and select Component Wizard option at the bottom of the context menu. After the Welcome panel of the Component Wizard is displayed, select the "Let me select a type..." option and click Next to display the Component Type panel (Figure 14-5).

The Component Wizard can also be launched from the components view under Advanced Views. It can also be launched from the loser left panel in the Files view under Step 2.

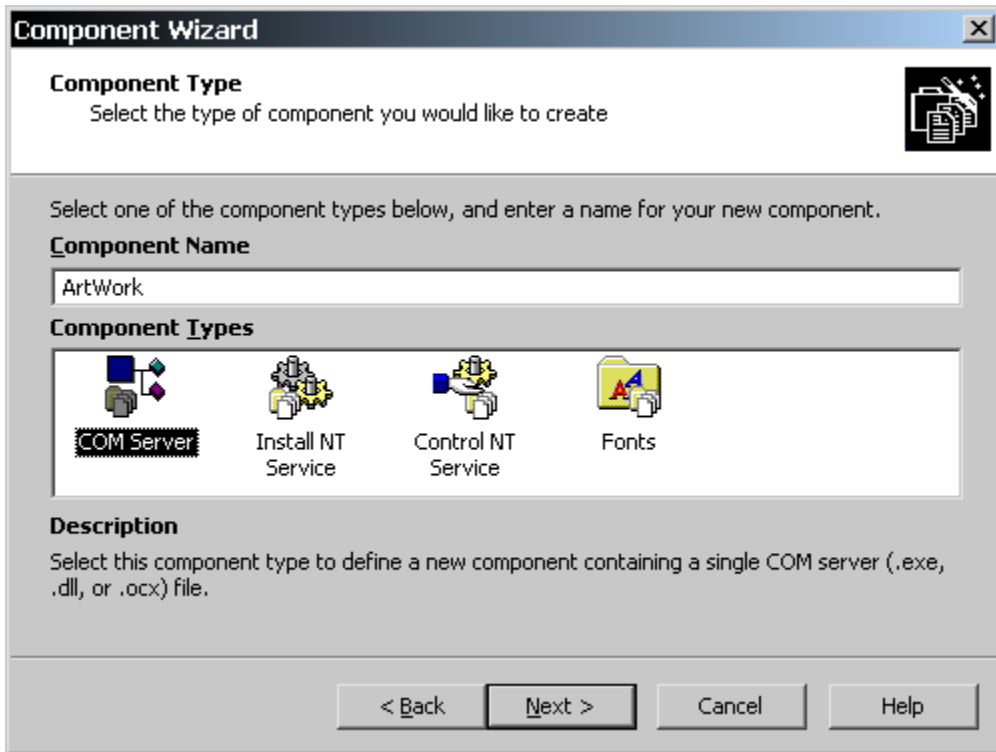


Figure 14-5: *The Component Type panel in the Component Wizard.*

In Component Type dialog, make sure that the COM Server component type is selected and enter the name of the component that you want to create. In this case, use the name ArtWork as shown in Figure 14-5. Click Next to display the Destination panel (Figure 14-6).

The Destination panel is where you select the destination where the component is to be installed. The dialog contains the familiar drop-down menu of predefined locations. In the case of the ArtWork component, leave the destination as [INSTALLDIR] and click Next. This will display the COM Server-Destination dialog as shown in Figure 14-6.

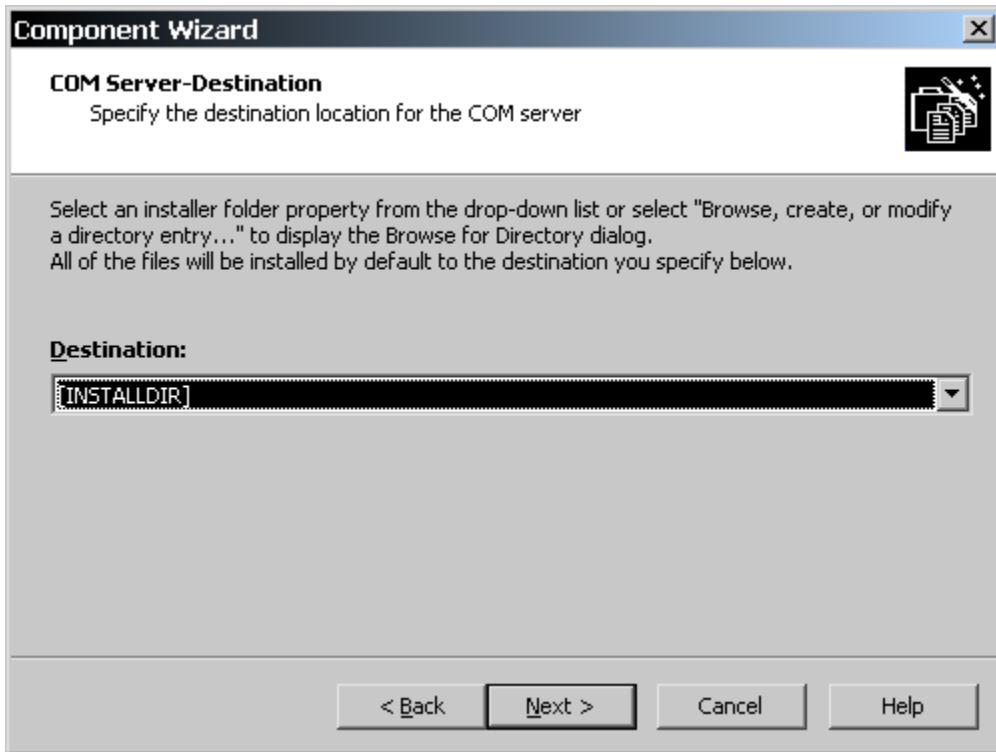


Figure 14-6: *The COM Server-Destination panel in the Component Wizard.*

The COM Server File dialog is where you identify the file that is the COM server (Figure 14-7). In this panel, you can browse for files with the extensions .exe, .dll, .ocx, or .tlb. A file with an .ocx extension is just a COM DLL that has a specified number of interfaces that it has to support. A file with a .tlb extension is a type library and this file contains information about the objects that a COM server exposes. Type libraries can be separate files with a .tlb extension or they can be resources contained inside the COM server itself. This is the case with the ArtWork.dll COM server.

In the COM Server File panel, you can select to have the Component Wizard extract the COM information automatically or you can elect to enter the COM registration information manually. The default is to have the information from the identified COM server extracted automatically. If the COM server is an executable, the second option is enabled and it is where you can specify that the COM server will run as an NT service. For this example, you will use the manual approach for entering the

COM registration information so you should deselect the “Extract registration information” check box and click Next.

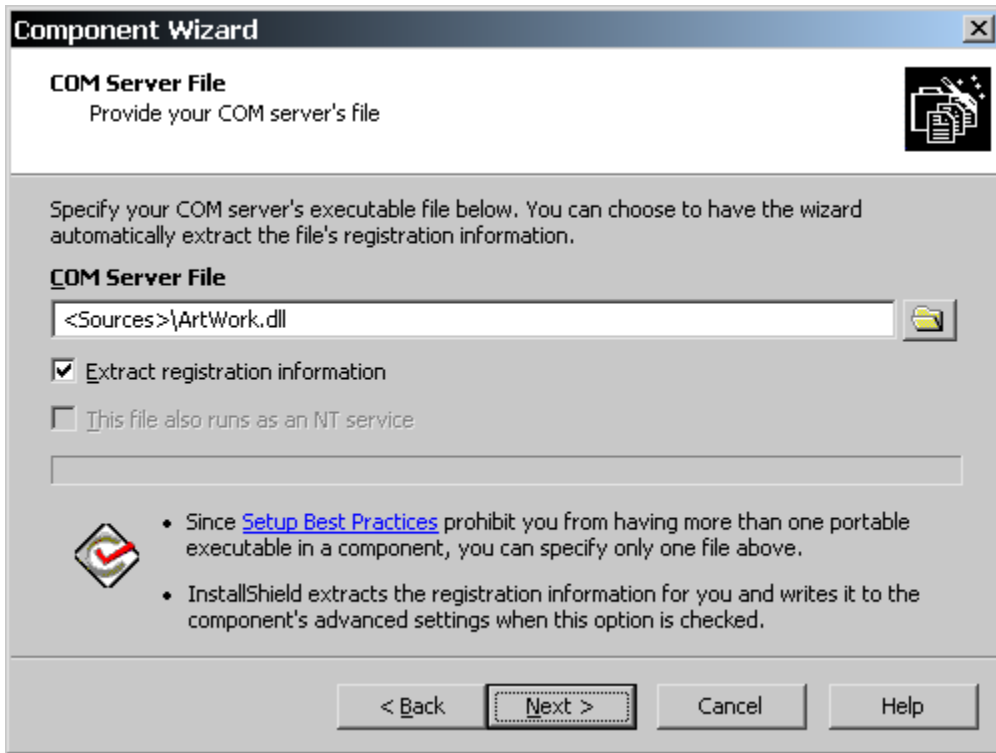


Figure 14-7: *The COM Server File panel in the Component Wizard.*

The Classes panel is next (Figure 14-8). You need to enter exactly what is shown in Figure 14-8 for the ArtWork component to be registered correctly. To create the ProgID, click the Add button and then press the F2 function key. This allows you to modify the default name of the ProgID.

Enter the GUID shown into the ClassId edit field and the string shown in the Version-Independent ProgID edit field. The GUID uniquely defines the COM class implemented by ArtWork.dll and the ProgID is another method for referring to the class that is implemented by the COM server. The third part of the ProgID is the version number that refers to the class implemented by a particular version of the COM server. The Version-Independent ProgID is way to have a constant reference to the latest version of the class.

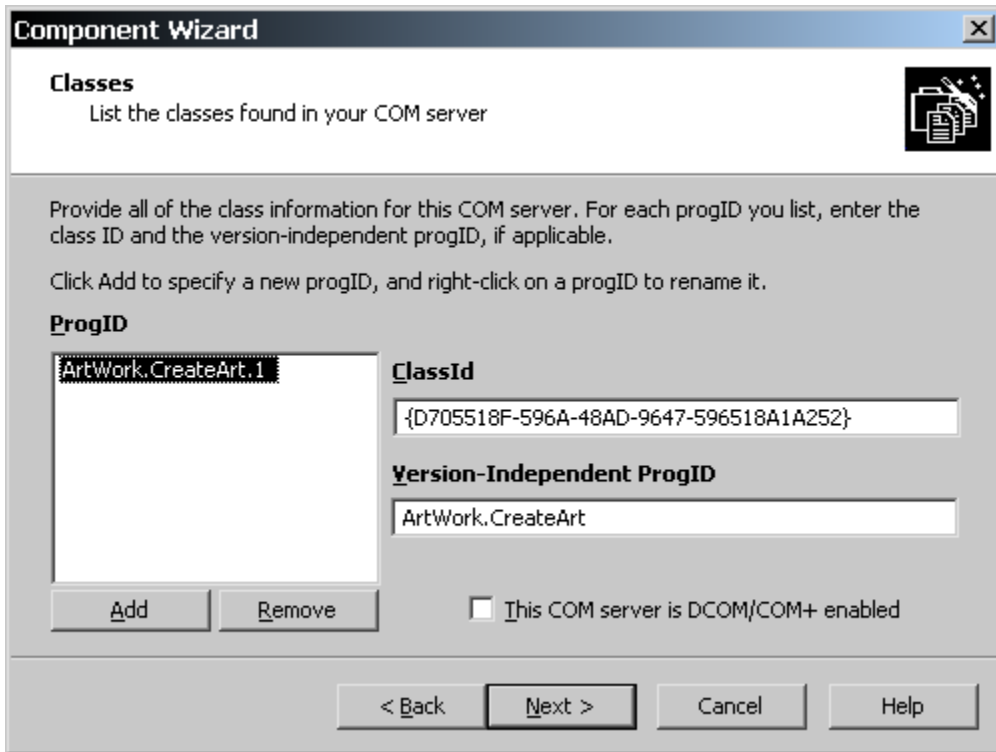


Figure 14-8: *The Classes panel in the Component Wizard.*

After making the entries shown in Figure 14-8, click Next to move to the Context Types panel (Figure 14-9). This is where you can define whether the COM server will run in the same process as the client or whether it will run in a separate process. Since the COM server is a DLL and it is a 32-bit DLL, select the InprocServer32 context type. If the COM server were a 32-bit executable then you would select the LocalServer32 as the context type. There are also InprocServer and LocalServer context options that can be selected but those would only be appropriate for 16-bit COM servers.

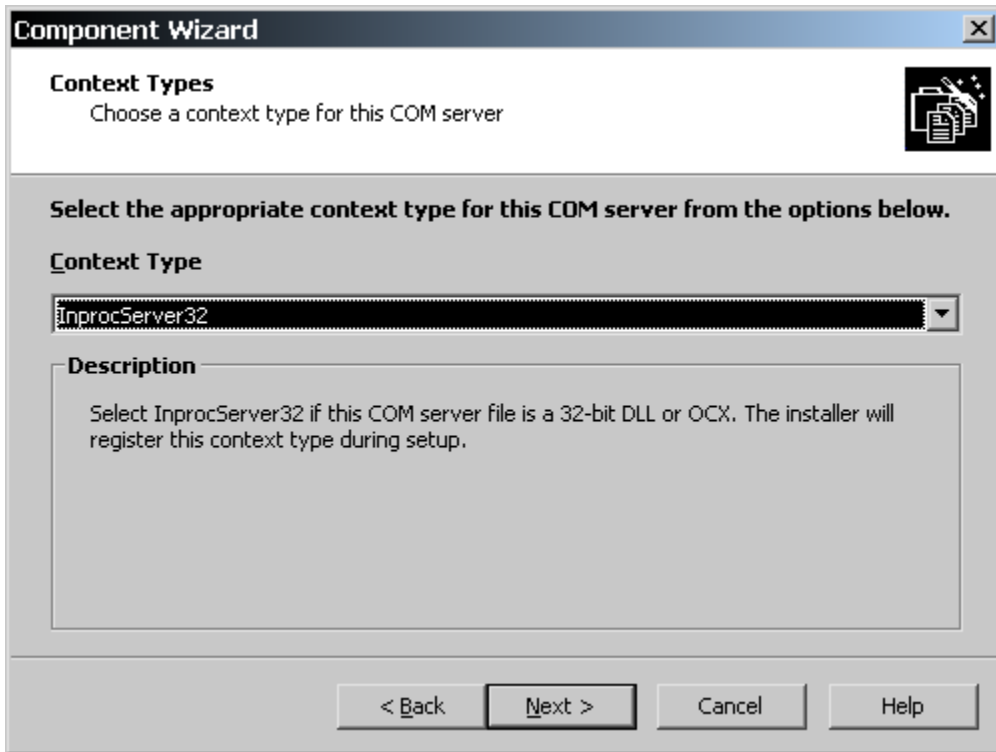


Figure 14-9: *The Context Types dialog in the Component Wizard.*

Click Next to move to the Type Library panel (Figure 14-10).

For the Type Library panel, you need to make the entries shown in Figure 14-10. The Type Library Description edit field is where you place the readable name to describe the type library. This is not a required field but it is important if the type library is going to be referenced by an application such as Visual Basic. The Type Library GUID edit field is necessary and the value entered here has to be as shown in Figure 14-10.

The Language edit field is also a required value. In this field, enter the language ID of the language supported by the type library. If the type library is language independent, type the value 0 in this field. The final field in this panel is the version of the type library. This is not a required field, but it is good to place the version number here. This version number consists of two 1-byte numbers separated by a period. For the ArtWork.dll COM server, enter the value 1.0.

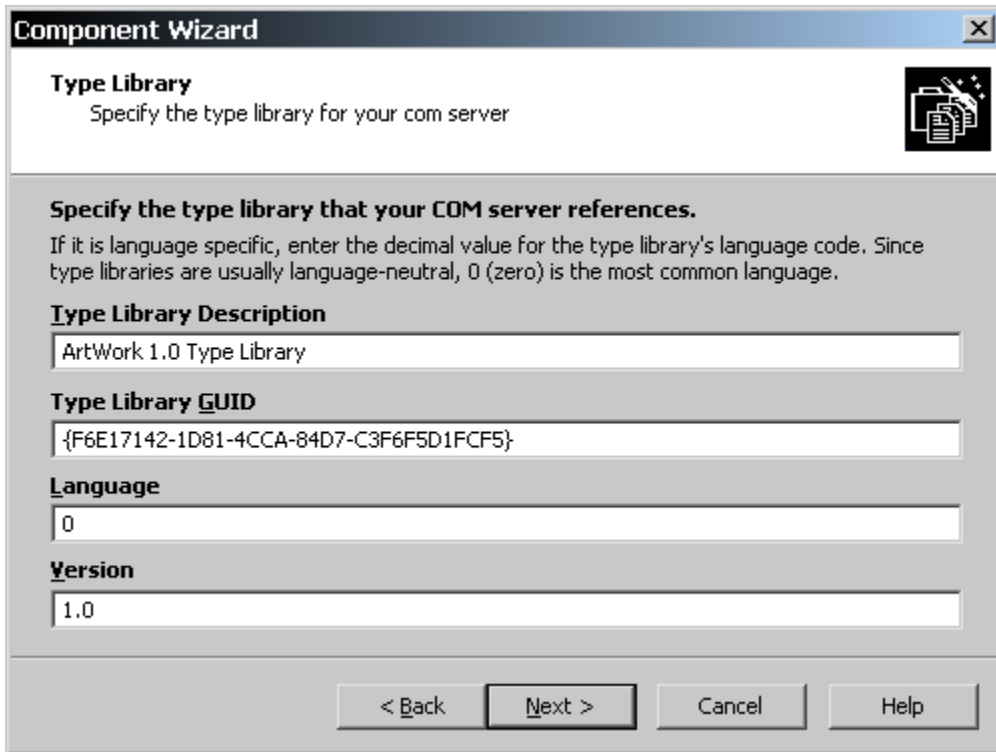


Figure 14-10: *The Type Library dialog in the Component Wizard.*

Click Next to move to the Summary panel. After reviewing your selections in the Summary panel, click Finish to create the ArtWork component under the MainProgram feature.

Until now, every time you built this project, the COM registration has been extracted during the build and placed in the various database tables. The COM registration information has never been saved in the project file itself. Performing the extraction of COM information during the build is the proper way to do it if the COM server is still under development, but to extract COM information at build time slows the build process. By running the Component Wizard in the previous example, you have placed the COM information in the project file. If you go to the ArtWork component in the Setup Design view, you should see something like what is shown in Figure 14-11.

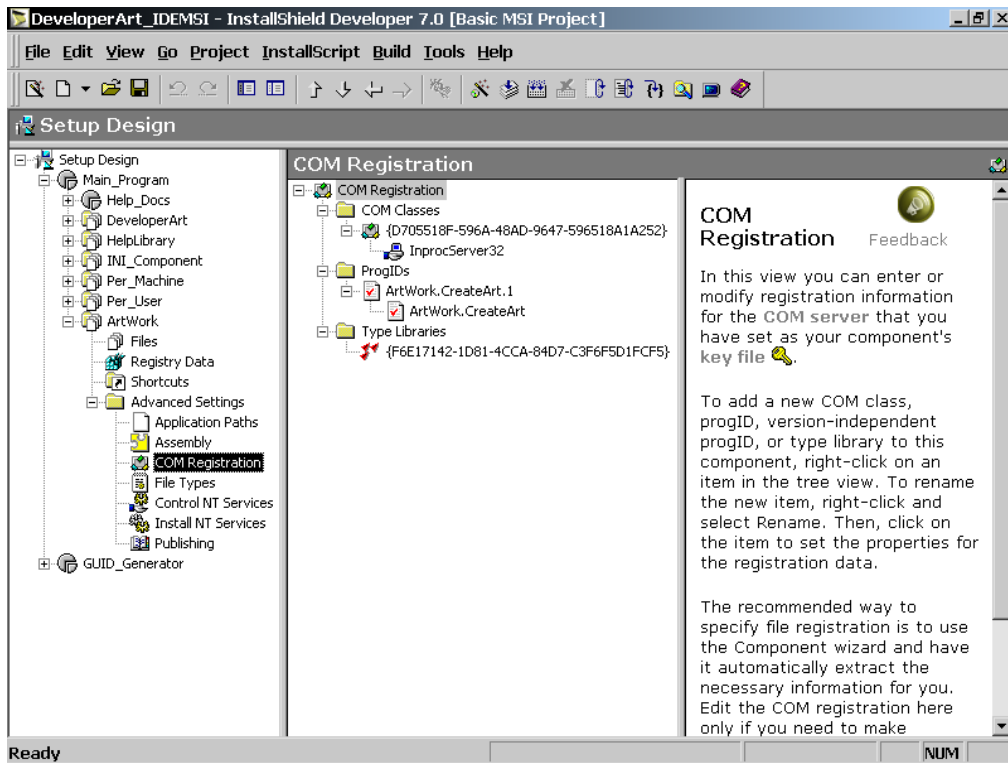


Figure 14-11: *The COM Registration view for the ArtWork component.*

In Figure 14-11 you can now see icons for the COM Classes, ProgIDs and Type Libraries. Except for the InprocServer32 entry under the Class GUID, each of the icons shown in Figure 14-11 has an associated property page where you are able to modify what was entered into the Component Wizard or enter values that were not part of the input requested by the Component Wizard. One thing that you should do is to enter a string for the Description property for the Class GUID and the two ProgIDs. You can use the same string for all three descriptions. A valid description for the Class GUID is shown in Figure 14-12.

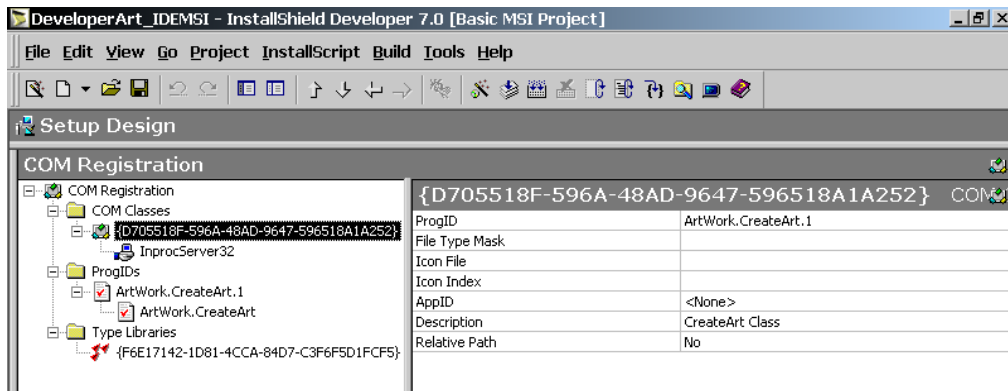


Figure 14-12: *The property page for the Class GUID.*

Now all you have to do is build the project and run the installation. When you run the application you will see that it runs and performs its drawing functions. This tells you that the ArtWork.dll COM server was registered correctly.

The next section discusses how to create components that install NT services.

NT Service Components

An NT service is a background task that runs on an NT/2000/XP platform. The code for this background task is housed in an executable (.EXE) and this executable can contain more than one NT service. All services that are contained in a single executable run as separate threads in the executable's process space. The term NT service can refer to a hardware driver service, a file system driver service, and a Win32-based service. This section addresses the subject of Win32-based services.

The NT Service Environment

There are five separate entities that come into play when you run an NT service. These are the service itself, the Service Control Manager (SCM), the Service Control Manager database, a service control program (SCP), and optionally a client application that is connected to the NT service. Figure 14-13 diagrams these five entities and their relationship.

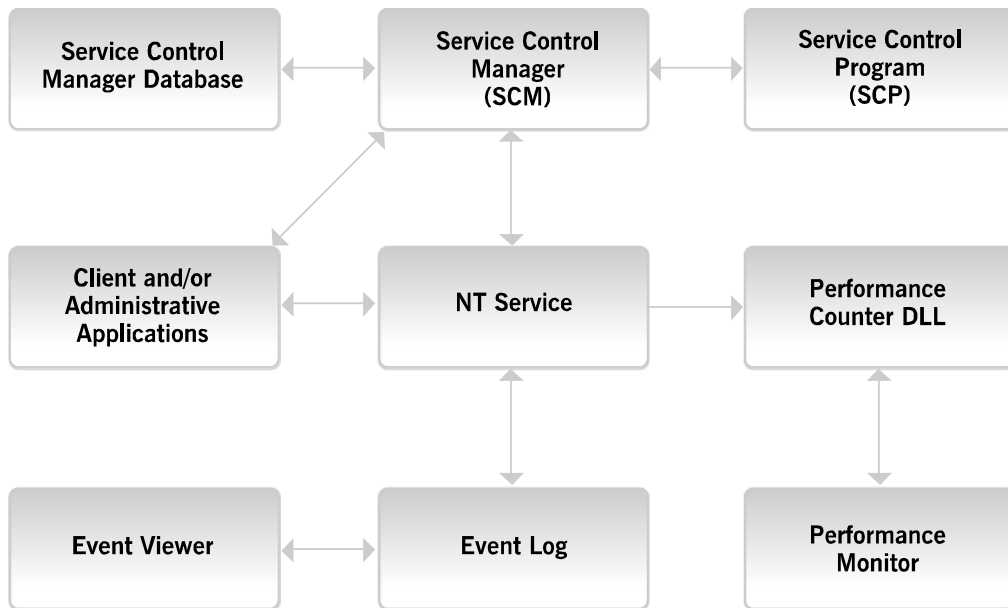


Figure 14-13: *The interaction of an NT service and its environment.*

In Figure 14-13, the arrows indicate the lines of communication between the service process and the surrounding environment. All control communication with the service is through the Service Control Manager. The other communication between the service and the environment relates to information that is passed to and from the service, relative to the work that the service is doing or the information that the service is sending to either the event log or the performance-monitoring facility. Below is a definition of each of these components shown in Figure 14-13.

The NT service: As stated, an NT service runs in a background process. Typically this process is a console application and does not have any interaction with the user. However, it is possible to allow an NT service to interact with the desktop. The structure of an NT service process is different than the normal console application because it is controlled by something called the Service Control Manager that is found on NT/2000 operating systems. The NT service and the Service Control Manager communicate with each other with the Service Control Manager sending control commands to the service and the service telling the Service Control Manager about its status.

The Service Control Manager (SCM): The SCM is an operating system component that is installed when either Windows NT or Windows 2000 is installed. The SCM is launched when the operating system is booted and is not shut down until the operating system is shut down. It is through this OS component that an NT service is told to start, stop, pause, or continue.

The Service Control Manager Database: The SCM maintains a database of all installed services. The persistent form of this database is a portion of the registry. There is also a version of this database that is held in memory that tracks the status of running services. When an NT service is installed, entries are made in this database.

Service Control Programs (SCP): A Service Control Program is a utility application, normally with a user interface, that permits a user to control the operation of an NT service. The normal control operations are start, stop, pause, and continue. The SCP that most people are familiar with is the Services applet in the Control Panel.

Event log and event viewer: Since most NT services run without any visual communication with the outside world, there needs to be a method for a service to report its status and/or errors that may occur. This method involves writing to the event log and using an event viewer to view what is in the event log. All services need to write to the event log to report any significant event that has occurred during its operation.

Performance counter DLL and performance monitor: For any real-world service, there needs to be a way for an administrator to see how the service is performing. Knowing the performance statistics for a service allows the administrator to tune the service for more efficient operation. Capturing the performance data of an NT service is the function of a performance counter DLL and looking at this performance data is the function of a performance monitor.

The client/administrative applications: A client application is the application for which the service is performing work. A client application can also perform the duties of a service control program. An administrative application is used to help configure and tune an NT service and most major services come with their own administrative application. Typically these administrative applications are

delivered as either a control panel applet or a Microsoft Management Console snap-in.

Now that the various components that come into play when running an NT service have been defined, we will take a closer look at each one of these in the following sections. Before that, we will take a look at the different types of services that can be created. It is important to know the type of service that is being installed so that the correct entries can be made in the Service Control Manager database.

Types of NT Services

Services fall into two major groups, those that are considered device drivers and those that are considered Win32-based services. This section provides a little more detail about the various types of NT services:

Kernel mode drivers: This is a 32-bit modular component that runs in kernel mode and, as such, has unrestricted access to the operating system and essentially can do what it wants. These drivers form the interface between the hardware and the rest of the operating system. Kernel mode drivers are normally installed into the %SystemRoot%\system32\drivers folder and they would have a .sys extension. The driver is entered into the SCM database just as is done for Win32 services.

File system drivers: On NT/2000/XP systems there does not have to be a resident file system. File systems are provided through installation and registration of file system drivers. Just as with kernel mode drivers, file system drivers are also installed to the %SystemRoot%\system32\drivers folder and registered in the SCM database. File system drivers then register themselves with the operating system's I/O Manager. More than one file system driver can be active at a time. When a volume is mounted or a remote name is being resolved, the I/O Manager calls all the registered file system drivers in turn until one of them recognizes the volume structure or remote name.

Win32 services that do not share a process: This type of service runs in user mode and does not have anything to do with the proper functioning of the operating system itself. In particular this type of service is the only service that is contained in the service executable. This means that there are only two threads running in the service process.

Win32 services that do share a process: This type of service runs in user mode and does not have anything to do with the proper functioning of the operating system itself. In particular this service executable there is potentially more than one service running. This means that there are more than two threads running in the service process when all services are started.

Win32 services that interact with the desktop: This type of service runs in user mode and does not have anything to do with the proper functioning of the operating system itself. A Win32 service that interacts with the desktop means that the service can display a user interface and permit interaction with the user. If there is more than one service in the same executable, then all services in the executable have to be set to interact with the desktop.

It is interesting to note that even though both kernel mode drivers and file system drivers run in kernel mode, they are treated as NT services from user mode.

Inside the Service Process

An NT service is implemented inside an executable and there can be any number of services implemented in a single executable. Each NT service that is implemented runs in a separate thread of the service process that is created by the executable when it is launched. As shown in Figure 14-14, there is a minimum of two threads running even for an executable that houses only one service.

The first thread in a service process is always the one that contains the entry point for the service executable and is the thread that communicates with the Service Control Manager. The first thread in a process is always thread 0. Each service runs in a thread numbered from 1 on up. Thread 0 contains a handler function for each service thread that is used to pass control requests between the Service Control Manager and the service thread.

The possible control requests that can be sent by the Service Control Manager to a service via its handler function are Stop, Start, Pause, Continue, and Interrogate. The service responds directly to the Service Control Manager with regard to its present state. The types of states that a service can be in are Stopped, Running, Paused, Getting Ready to Stop, Getting Ready to Start, Getting Ready to Pause, and Getting Ready to Continue. If the service is part of a client/server application, the client communicates directly with the service without going through the handler function.

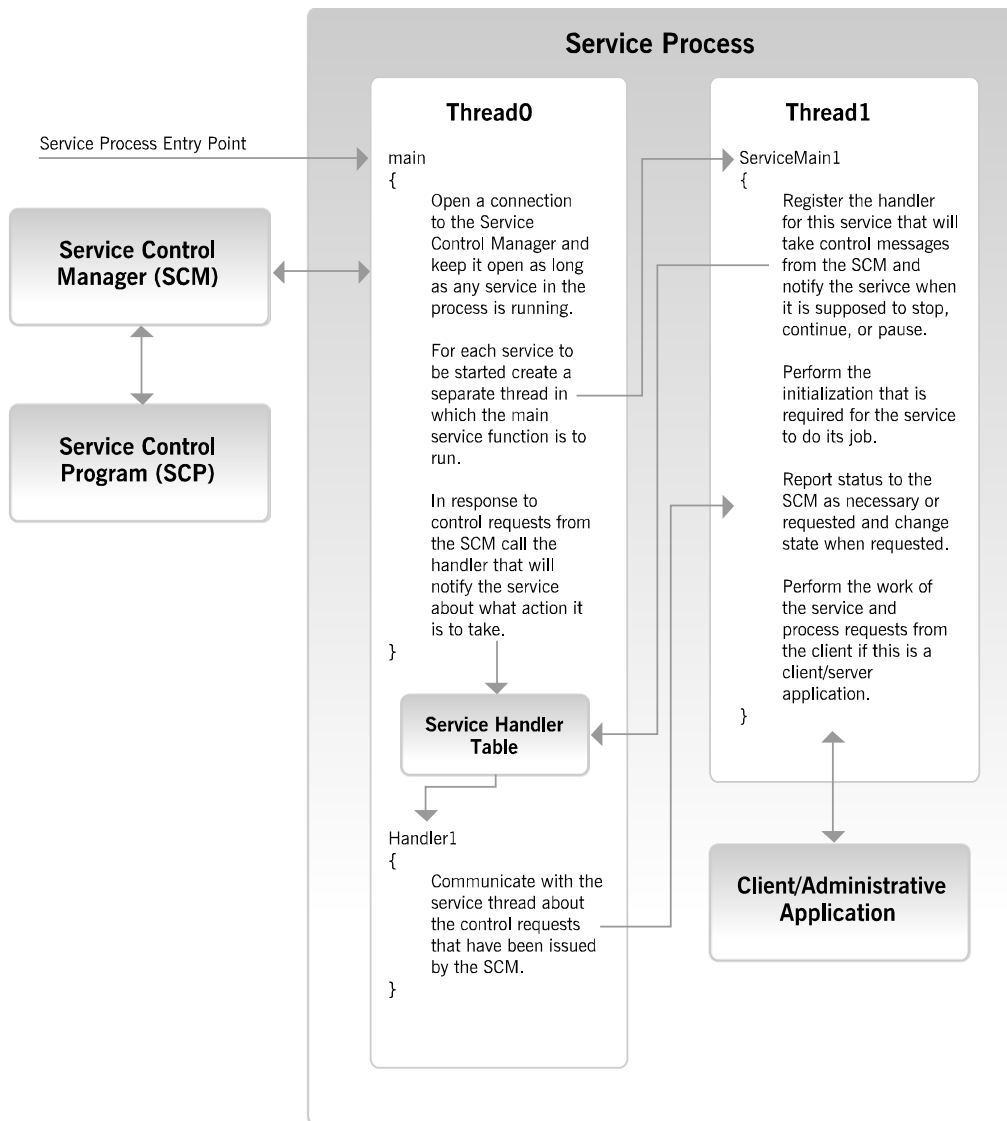


Figure 14-14: *The basic mechanism of running an NT service.*

An executable that houses an NT service cannot just be launched in order to start the service. On Windows NT, Windows 2000, and Windows XP, it is necessary to tell the operating system that a particular executable needs to be treated as an NT service. This is the topic of the next section.

The Service Control Manager Database

Windows NT-based operating systems recognize an executable as containing an NT service only if the service is registered in the Service Control Manager database. This database is a section of the registry that is located under the following key:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services
```

Under this registry key, there is a sub-key for each installed NT service and the name of the key is the name of the service. The values that are written against this sub-key provide the data that is required to find, start, configure, and stop the service. There can be sub-keys under each service key and these can be used to hold values that are private to the service itself.

The Service Control Manager database is always backed up by the operating system. If a boot of the system is successful, a copy of the current database is copied to a registry key that has the following format:

```
HKEY_LOCAL_MACHINE\SYSTEM\ControlSetXXX\Services\
```

In the above key, the string XXX represents a numerical value of the last known good configuration and this value is stored in the following registry key and named value:

```
HKEY_LOCAL_MACHINE\SYSTEM\Select\LastKnownGood
```

If any changes made to the active system configuration cause a failure, at the next boot, the system restores the Service Control Manager database using the copy specified by the LastKnownGood registry value.

It should be clear now that the real work of installing, controlling, and removing an NT service revolves around the manipulation of the Service Control Manager database.

Installing, Controlling, and Removing an NT Service

To install an NT service, the Windows API `CreateService` is used. This function takes a number of arguments and these arguments are used to write the values against the service sub-key. The Windows Installer database contains a `ServiceInstall` table that holds the values for the arguments to the `CreateService` Windows API. The Windows Installer standard action `InstallServices` reads the rows in the `ServiceInstall` table and then passes these values to the `CreateService` API.

Once a service is installed, there are a number of actions that can be taken to control it. The standard control requests, as listed earlier, are `Start`, `Stop`, `Pause`, `Continue`, and `Interrogate`. The two control requests that we are interested in as far as installation programs are concerned are the `Start` and `Stop` requests. To start a service, the `StartService` Windows API is used. The `StartServices` standard action in the Windows Installer reads the values in the `ServiceControl` table and then passes these values to the `StartService` API to start a service that has been installed.

An NT service is stopped by using the `ControlService` Windows API and passing it a control code that specifies stop. In the Windows Installer, the `StopServices` standard action reads the `ServiceControl` table and passes the values in this table to the `ControlService` API.

Using the `DeleteService` Windows API uninstalls an NT service. This API does not take action to delete the file itself, but only removes the entries in the Service Control Manager database for the specified service. The Windows Installer `DeleteServices` standard action reads the values in the `ServiceControl` table and passes these values on to the `DeleteService` API.

You should notice that in the cases of starting, stopping, and deleting a service, the required information is contained in the `ServiceControl` table. The `Event` column in this table informs the standard actions that read this table whether there is any action to take. For example, if there is only one row in the `ServiceControl` table and the `Event` column indicates that a service is to be started then only the `StartServices` standard action performs any operation. The other standard actions that read this table see this one row as a no-op.

Using the Component Wizard to Install and Control an NT Service

As we work through the various dialogs of the Component Wizard to create the components for installing and controlling an NT service, further details of NT services will be discussed. This is necessary so you will know why the Component Wizard requires certain information. Once again the DeveloperArt_IDEMSI project will be used for this example, but the same operations can be performed in the DeveloperArt_IDEStd project.

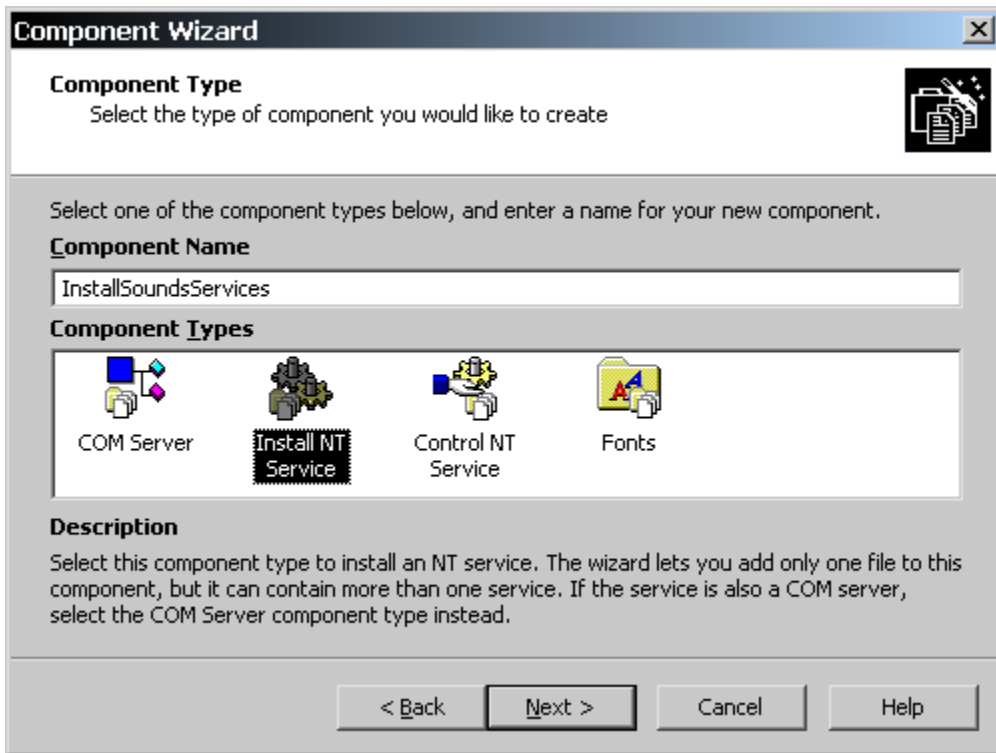


Figure 14-15: *Naming the component for installing an NT service.*

The included CD-ROM contains an executable Sounds.exe that houses two NT services. You will use this executable for this example. These two services play .wav files in sequence and the two services communicate with each other so that they

alternate in playing the two sounds. Any two wave files will work as long as they are named FirstSound.wav and SecondSound.wav. The .wav files that are on the CD-ROM are the logon and logoff sounds used by Windows 2000. You need to have a sound card to be able to hear these sounds.

Open the DeveloperArt_IDEMSI project, go to the Setup Design view and create a new top-level feature named SoundsService. All components that you create in this example will be placed under this SoundsService feature. Then right click on the SoundsService feature and select the Component Wizard option. In the Welcome panel, select the “Let me select a type...” option and click Next.

The first component that you will create is one that will install the two services. Then you will create two more components that will provide the logic for how you want to handle the NT services after they are installed, but before the installation is complete.

In the Component Type panel (Figure 14-15) create a component named InstallSoundsServices. Type InstallSoundsServices in the Component Name field and select Install NT Service in the Component Types selection box. After the component name, all of the information collected by the Component Wizard is used to populate the ServiceInstall table. Click Next to move to the NT Service Executable panel.

The entries to be made in the NT Service Executable dialog are shown in Figure 14-16. First, browse for the file Sounds.exe. You must name the two services that are housed inside Sounds.exe. Click the Add button to add and name the services, as shown in Figure 14-16. You can use the F2 function key to edit the default name for the service. The names of the services provided in this panel are used to register the services in the Service Control Manager database and these names are also used by the Service Control Manager to open the service. The names of a service need to be what are used inside the service. These names cannot be longer than 256 characters.

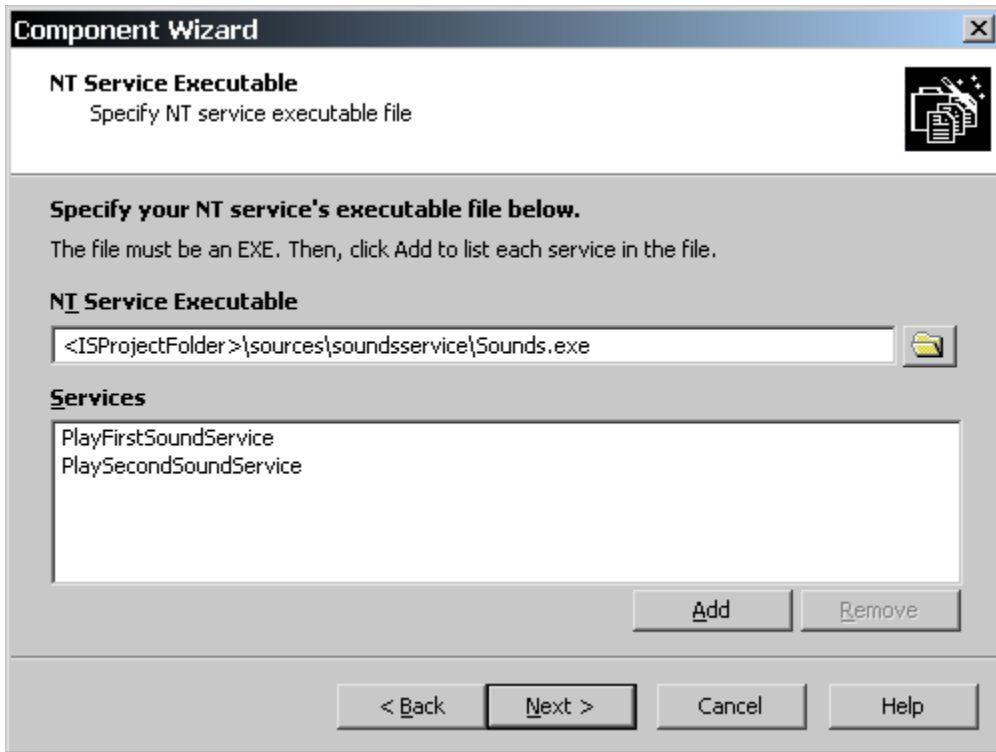


Figure 14-16: *The entries for the NT Service Executable panel.*

In the Service Type Information panel (Figure 14-17), you define the service type and the display name for each of the services that were defined in the NT Service Executable panel. For each service being installed you want to give it a display name that will be used in the Windows NT Services Control Panel applet or in the Services Microsoft Management Console snap-in found in Windows 2000 and Windows XP. This display name is also limited to 256 characters. Click Next to move to the Service Type Information dialog. This dialog is shown in Figure 14-17.

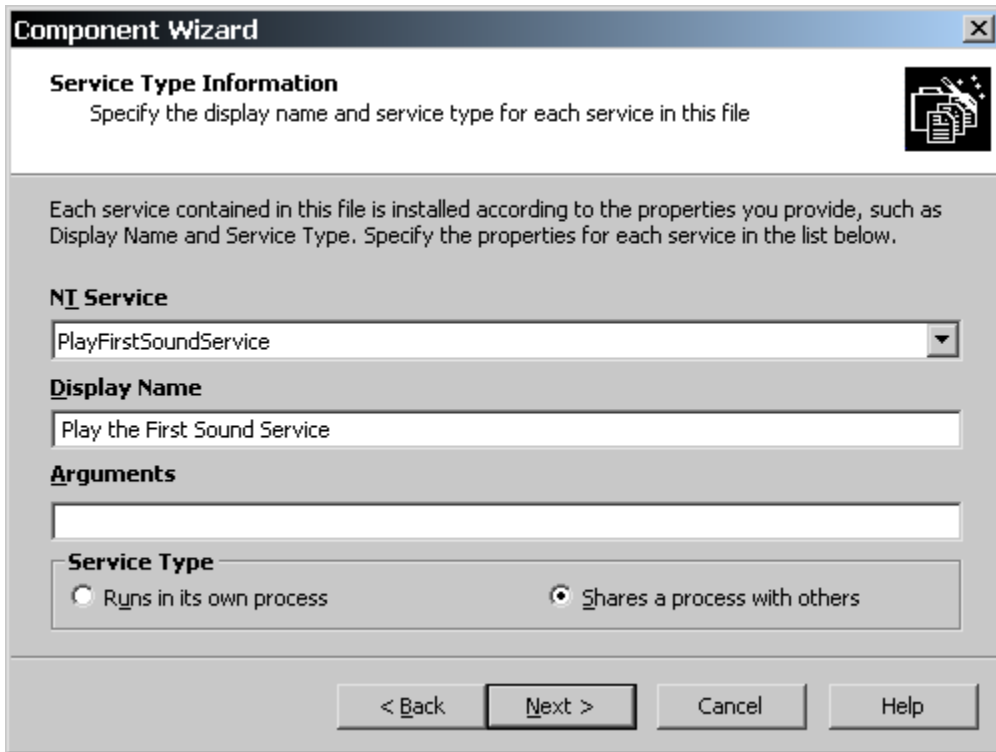


Figure 14-17: *Entering the information that describes the services being installed.*

To provide a display name for each of the services being installed, select the service from the drop-down NT Service menu and then enter a display name for the service in the Display Name field. For each service you also have to identify the service type, which you do by selecting one of the radio buttons in the Service Type selection box. Since you are installing an executable that contains more than one service, you must select the “Shares a process with others” option. Both services must have this option selected.

Leave the Arguments edit field empty because neither service takes any arguments. If you refer to Figure 14-14, you will see that there is a main function and that each service has a ServiceMain function. The main function takes no arguments, but each ServiceMain function can take arguments. However, in this case, there are no arguments that need to be passed. There is no way for you to know what arguments need to be passed to any particular service unless the service’s developer tells you. Click Next to get to the Service Start Type Information dialog.

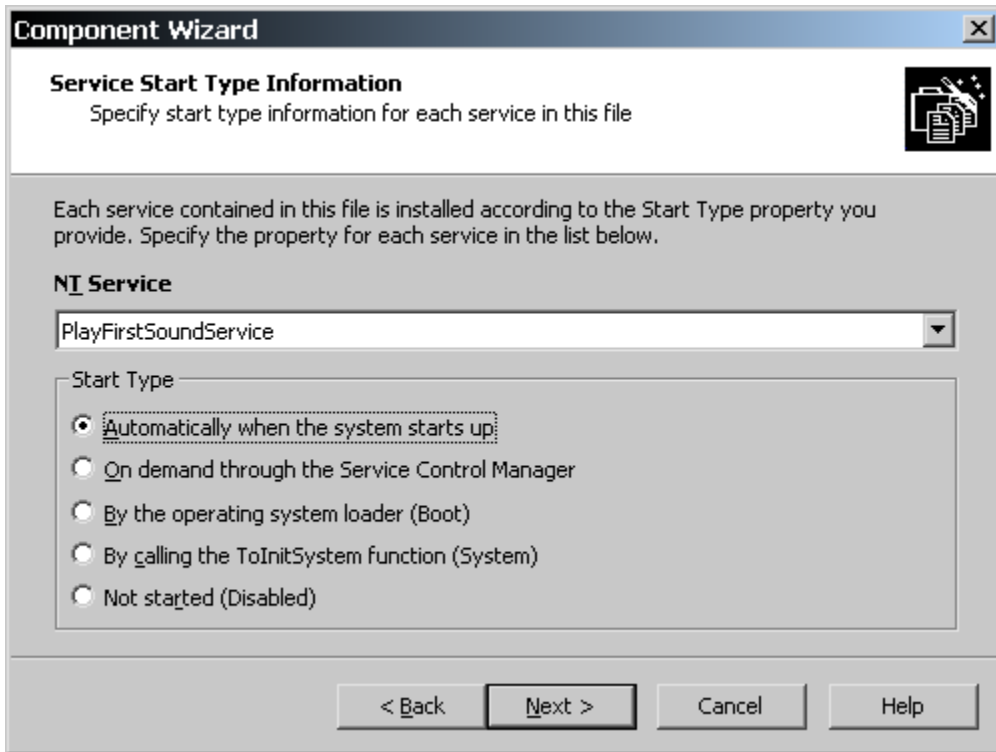


Figure 14-18: *Entering the start type information for an NT service.*

The Service Start Type Information dialog is shown in Figure 14-18. In the Service Start Type Information panel, you indicate the start type information for each of the services being installed. The meaning of each Start Type option is described in the following list:

Automatically when the system starts up: This option is applicable to Win32 services that need to be running before the user is presented with the logon dialog box. This ensures that all services that the user requires are running by the time the logon process is complete.

On demand through the Service Control Manager: With this option, a service is not started until an administrator starts it from the Services Control Panel applet or the Microsoft Management Console snap-in. Services with this start type can also be started if another service depends on it or if an application specifically starts it with a call to the `StartService` API.

By the operating system loader: This start type is applicable only to device drivers that are required in order for the operating system to boot.

By calling the IoInitSystem function: This option is also for device drivers and file system drivers that need to be loaded for proper system operation. These drivers are loaded prior to those services that have the automatic start type.

Not started: This type disables a service so that it cannot be started by the system. Any service that depends on a disabled service will not be able to start. Disabling a service is a method for uncovering the source of system problems.

For this example, select the “Automatically when the system starts...” option for both services. Do this by selecting each service being installed from the NT Service drop-down menu and setting the start type. Click Next to move to the Service Load Order panel (Figure 14-19).

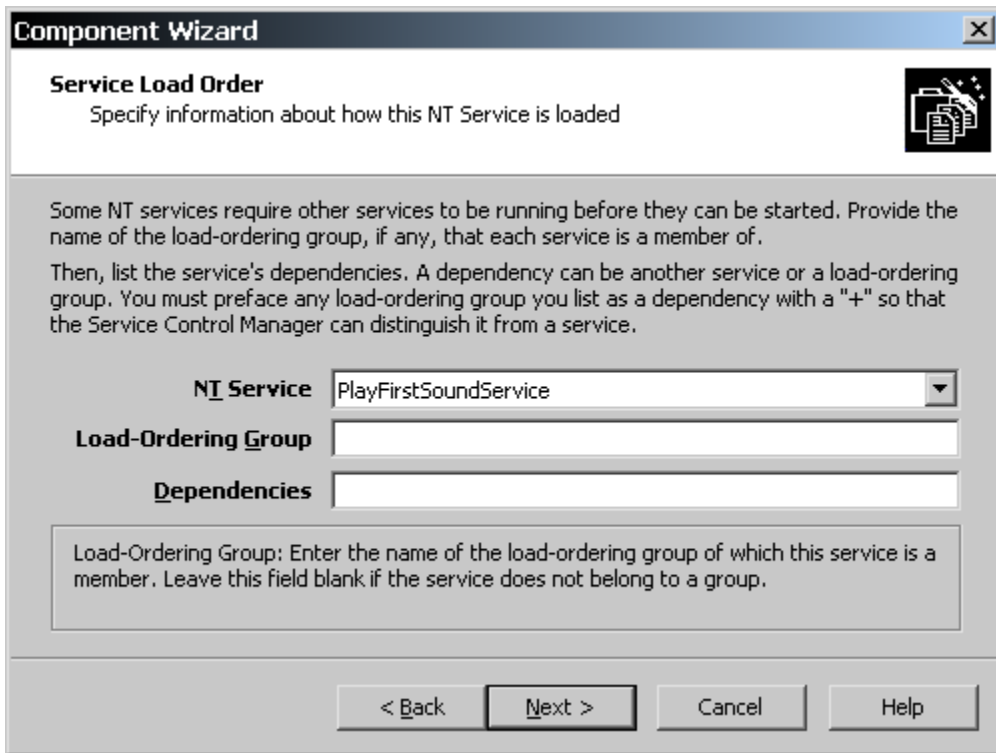


Figure 14-19: *The Service Load Order panel in the Component Wizard.*

The Service Load Order panel is used to specify any types of load ordering or dependencies required by the services being installed. For the services in this example, you do not have to enter any values in this panel.

We should discuss, however, the meaning of the two types of entries that can be made. The first topic is the purpose of a load-ordering group. A load-ordering group is a mechanism for grouping services together so that they are all loaded before services that do not belong to a group. System services are divided into groups and when the system boots it works through these predefined load-ordering groups to load the system services in the correct order. A list of the predefined load-ordering groups that are defined by the system can be found under the following registry key:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\GroupOrderList
```

The order shown for these load-ordering groups is not the order in which the members of these groups are loaded. The actual order in which these groups are loaded is defined by another registry key. This registry key is as follows:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder
```

There are three possibilities with load-ordering groups: you can identify your service to be part of one of the predefined groups, create a new group for your service, or not have it be part of a load-ordering group. It is probably not a good idea to assign a Win32 service to one of the predefined load-ordering groups because these groups are used to load operating system services. You could create a new group name. This would have the effect of loading your services after the services assigned to any of the predefined groups, but before any of the services that are not part of a group. Finally you could use the approach that most, if not all, Win32 services use and not specify a group. For example, the Windows Installer executable houses an NT service and it is not assigned to a group.

The next order of discussion with regard to the Service Load Order dialog is the specification of dependencies for a service. A dependency can be another service or it can be a load-ordering group. A dependency can also be a combination of services and load ordering groups. In the Service Load Order panel, you would add your dependencies using a comma delimiter and also place a plus sign (+) in front of any name that is a load-ordering group. The following rules are used when a service is identified to have dependencies:

- A service that depends on other services is not started until the Service Control Manager has started all the dependent services.
- If a service is defined to be of the “start on demand” type, then when action is taken to start the service, all dependent services are started first if they are not already started.
- A service with a load-ordering group dependency does not start until at least one of the services in the dependent load-ordering group has been started.

The next panel in the wizard is the Error Control panel (Figure 14-20). In this panel, you define for each of the services being installed the type of error handling that needs to occur if the service fails to start when the system is starting.

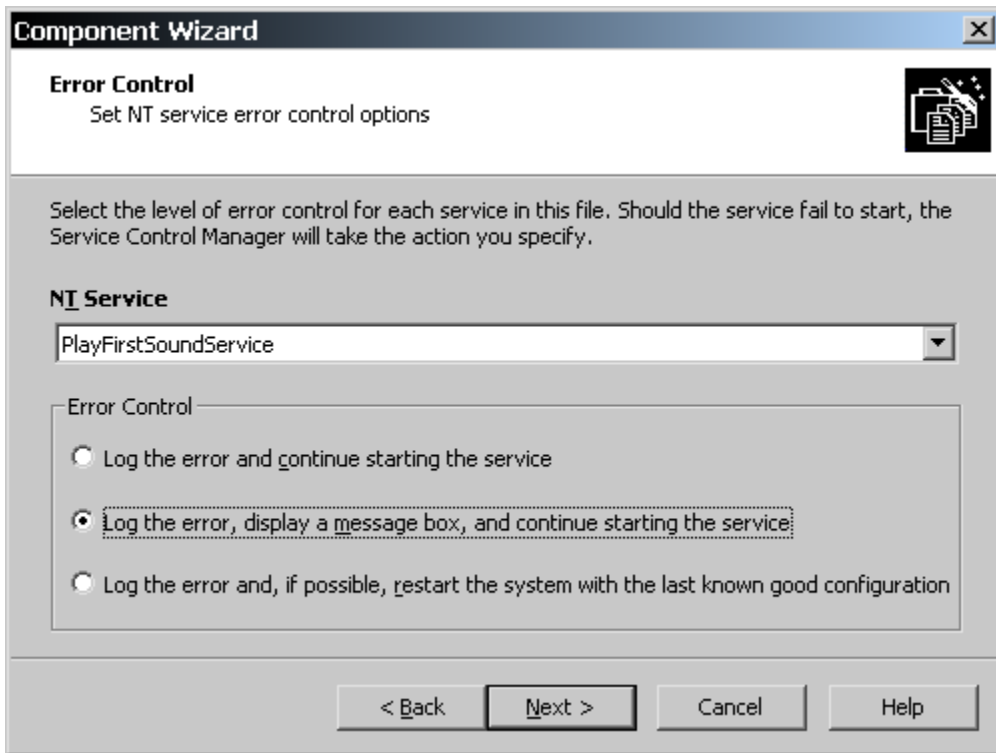


Figure 14-20: *Specifying how to handle errors if the NT services do not start.*

The Error Control panel presents three possible options for error control and these are discussed in the following list:

Log the error and continue starting the remaining services: The Service Control Manager logs the error in the event log and continues trying to start all the other services. The service that had the error is not started.

Log the error, display a message box, and continue starting the remaining services: The Service Control Manager logs the error in the event log and also notifies the user with a message box about the error. When the error message is dismissed, the Service Control Manager continues to start the remaining services. The service that had the error is not started.

Log the error and, if possible, restart the system with the last known good configuration: Here the Service Control Manager logs the error in the event log and then initiates a restart of the system using the last known good configuration in order to get the system started without interference from the service that does not start. If the error occurs when the last known good configuration is being used, the restart of the system fails.

If you are installing a Win32 service, there are only two Error Control options that are relevant for error handling. The two choices are “Log the error and continue starting the remaining services” and “Log the error, display a message box, and continue starting the remaining services.” It would be a very severe reaction to the failure of a Win32 service to start to mandate that the system be rebooted. The last Error Control option in the radio button group is appropriate only for device drivers that fail to start.

If you selected in the Service Start Type Information panel (Figure 14-18) to have the service automatically started when the system is started, then either of the first two error control options is acceptable. However, if you selected to have the service started on demand, you do not want to use the error control option where the Service Control Manager displays a message box. If the system is unattended, everything will stop until someone clicks the OK button on the message box. When using “start on demand” for a Win32 service, the only valid Error Control option is the first one.

Before you move on to the final input dialog for installing an NT service, we need to discuss one more aspect of error control. There is a special functionality provided by the Windows Installer that allows you to identify a service as being vital to the

installation of an application. A service that is designated as being vital terminates the installation if the service cannot be installed for some reason. This option cannot be selected in the Component Wizard, but has to be added by using the Direct Editor view. To identify a service as being vital, go to the ServiceInstall table in the Direct Editor and add the value 0x08000 to the value in the ErrorControl column.

In the Error Control panel, select the “Log the error, display a message box, and continue starting the remaining services” option and click Next to display the Service Logon panel (Figure 14-21).

There are two subjects worthy of discussion in this panel, interacting with the desktop and installing an NT service to a user account. Both of these topics are covered in the next two sections.

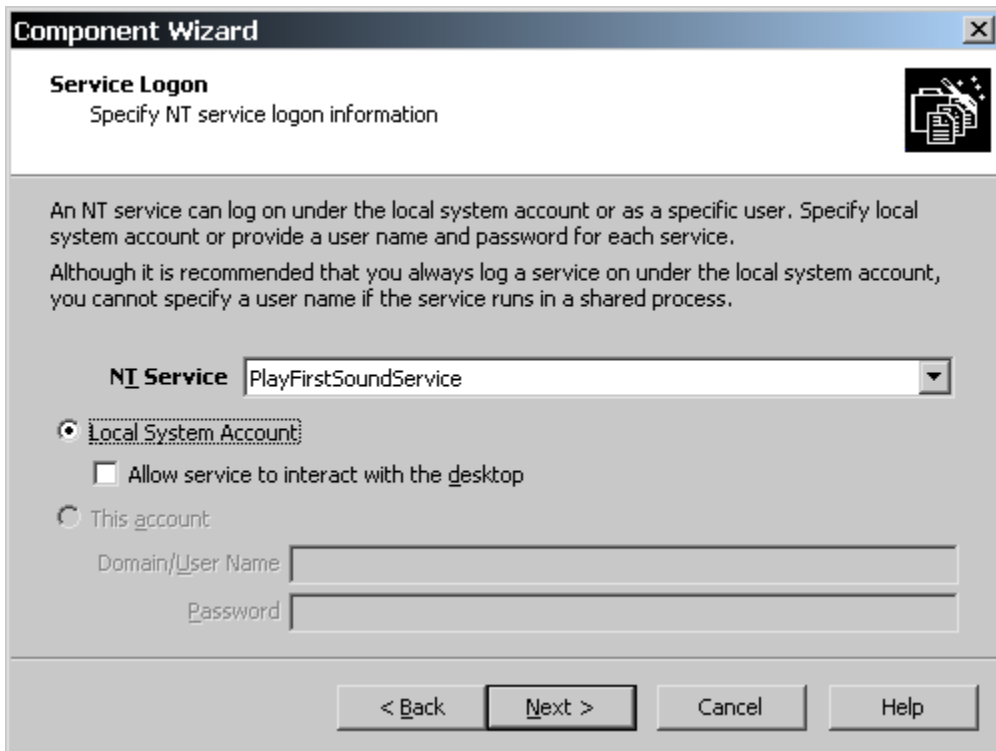


Figure 14-21: *The Service Logon panel in the Component Wizard.*

You do not have anything that you need to do in this panel since neither service needs to interact with the desktop, nor are you not going to install them to anything other than the local user account. Click Next to move to the Summary panel. Review in this dialog the actions that you have taken in the Component Wizard and then click Finish to create the component that will install the two services.

After the InstallSoundsServices component is created, go to this component in the Setup Design view and add the two .wav files to the component as shown in Figure 14-22.

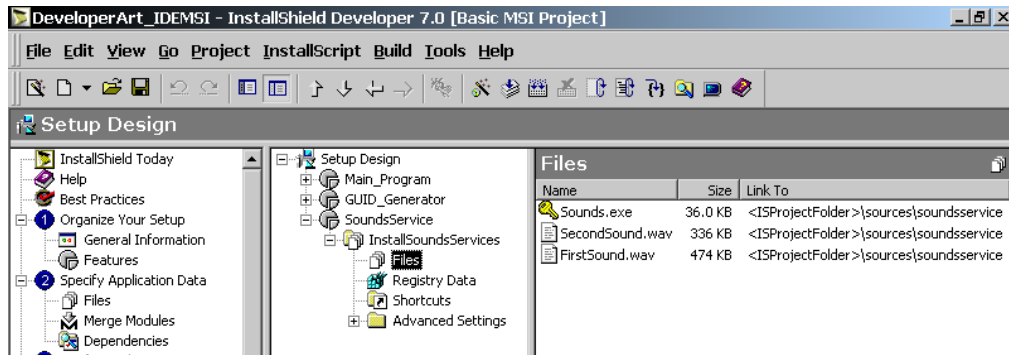


Figure 14-22: Adding the .wav files to the InstallSoundsServices component.

Next, open the Advanced Settings under the InstallSoundsServices component and click on the InstallNTServices icon. If you then click on the name of one of the services being installed by this component you will see a property page on the right of the screen where all the entries made in the Component Wizard are shown (Figure 14-23). You can modify or add items to any of these properties.

One of the properties that you want to enter a value for is in the Description field. This entry cannot be made in the Component Wizard. You can enter the same description for both services. This description will be shown beside the display name of the service in the Services Control Panel applet in Windows NT or in the Services Microsoft Management Console snap-in in Windows 2000.

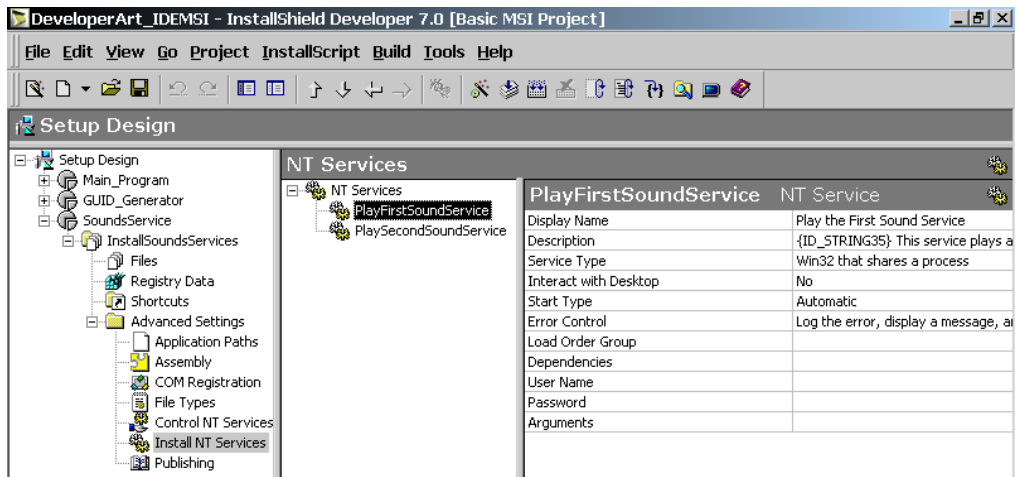


Figure 14-23: *The property page for the PlayFirstSoundService service.*

To finish creating the functionality for these services, you need to define how these services are to be treated after they are installed. If you did nothing, these services would not start until the next time you booted the system. What you need to do is use the Component Wizard to create two new components, one for each of the two services that are being installed, that define what to do with services as soon as they are installed.

You need to keep in mind that these two components will have nothing to do with the functioning of the services after the installation is complete. They will have some purpose during the uninstallation of the component that installed the services. This means that these two service control components need to be in the same feature used to install the NT services.

You start this process with the Component Wizard to create the two control components under the SoundsService feature as shown in Figure 14-24.

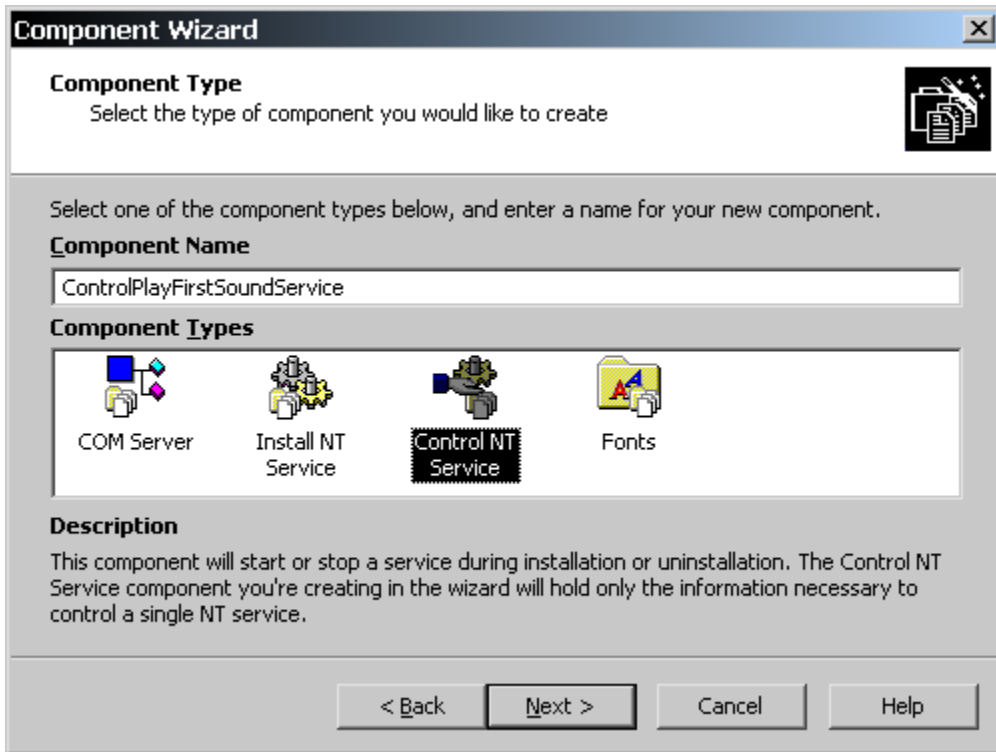


Figure 14-24: *Creating a component to control an NT service.*

After naming the control component for the PlayFirstSoundService service as shown in Figure 14-24, click Next to move to the Specify Service panel (Figure 14-25).

In the Specify Service panel, you can choose to control a service that is already on the target system or control one of the services that is being installed with the present installation. You want to control one of the services that are being installed with the present installation. Select the first name in the list box.

If you were performing an upgrade of a service that is running on the target system, you would want to stop that service from running so you could install the new executable. In this scenario you would then want to select the first radio button on the Specify Service dialog and name the service that you wanted to control. You would then create another component that controlled the service that would run from the upgraded executable.

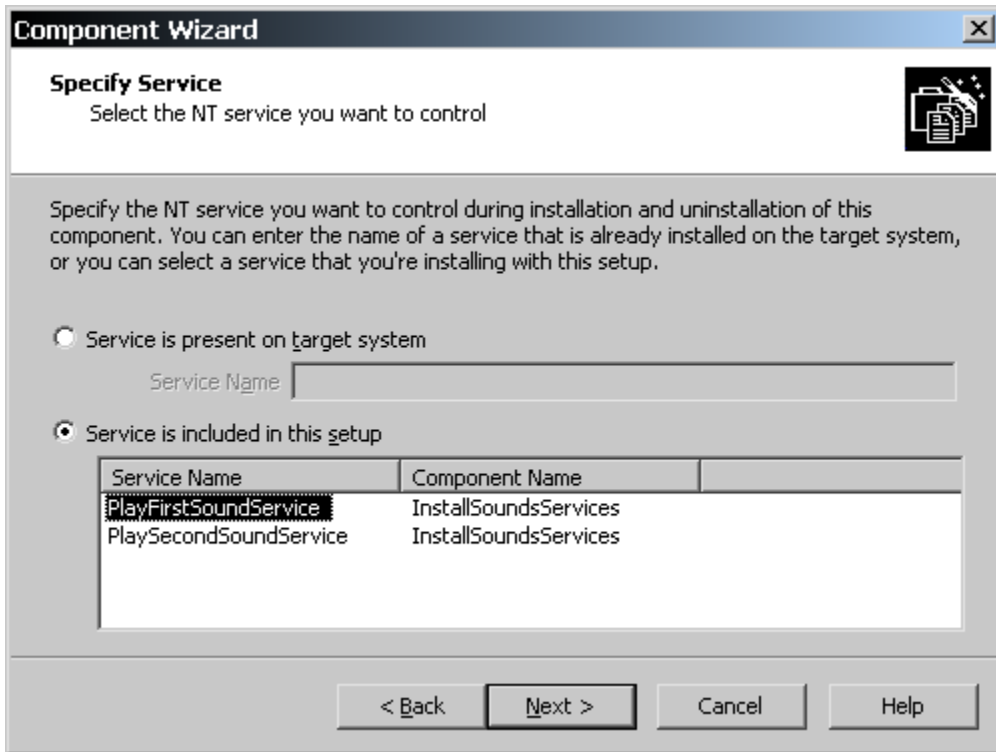


Figure 14-25: *Specifying the service to be controlled during the installation.*

In the next two dialogs in the Component Wizard, you will define what events are to be sent to the service via the Service Control Manager during installation and during uninstallation.

The Installation Events dialog (Figure 14-26) controls how you want the service started during the installation. For this example, make the entries as shown in Figure 14-26. In the Installation Events panel, you want to have the service started and, since none of the services being installed takes any argument, the Arguments edit field can be left blank. In the case of an upgrade, this is where you would select to stop and delete the service that was being upgraded. You would also make sure that you deselected the option to start the service. There is also the option to have no event occur during an installation.

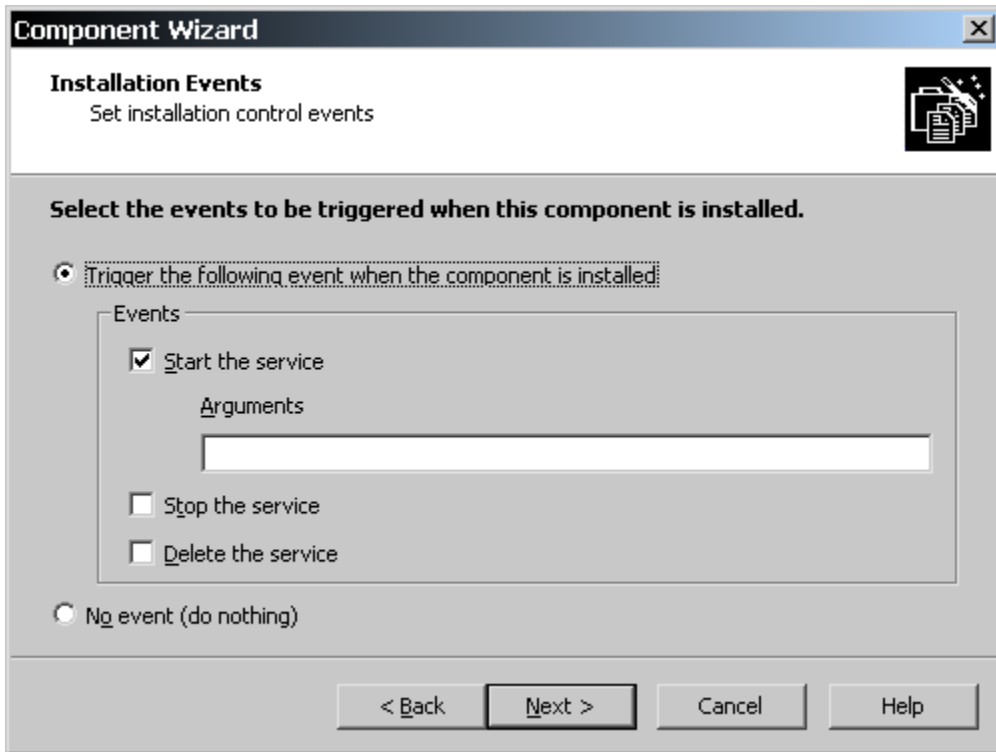


Figure 14-26: *Specifying the events to occur during the installation of an NT service.*

Click Next to move to the Uninstallation events dialog. In the Uninstallation Events dialog, you indicate what to do with the service when the application is being uninstalled.

Here you want to specify that the service be stopped and deleted when the application is uninstalled. Make sure that you deselect the “Start the service” option; otherwise, you will have an application that cannot be uninstalled without some special work on the cached package.



Figure 14-27: *Specifying the events to occur during the uninstallation of an NT service.*

The final input dialog for controlling a service is the Wait Type dialog (Figure 14-28). In this panel, you need to specify whether the Windows Installer is to wait until the events selected in the previous two dialogs are to complete before continuing or that the event has been started but not necessarily completed.

If it is critical to the completion of the installation or uninstallation that the event complete successfully, you would have the installation wait for the event to complete; otherwise, you can have the installation or uninstallation wait until each event has started but not completed to continue the operation.

For this example, make the selections shown in Figure 14-28. During the installation, it does not matter if the service starts, so you can select the second radio button in the Installation section. However, during an uninstallation, you probably want to wait until the Stop and Delete events have completed, otherwise an error in either of these

events could leave the machine in an unknown state if you decide to continue the uninstallation after you get the error message box.

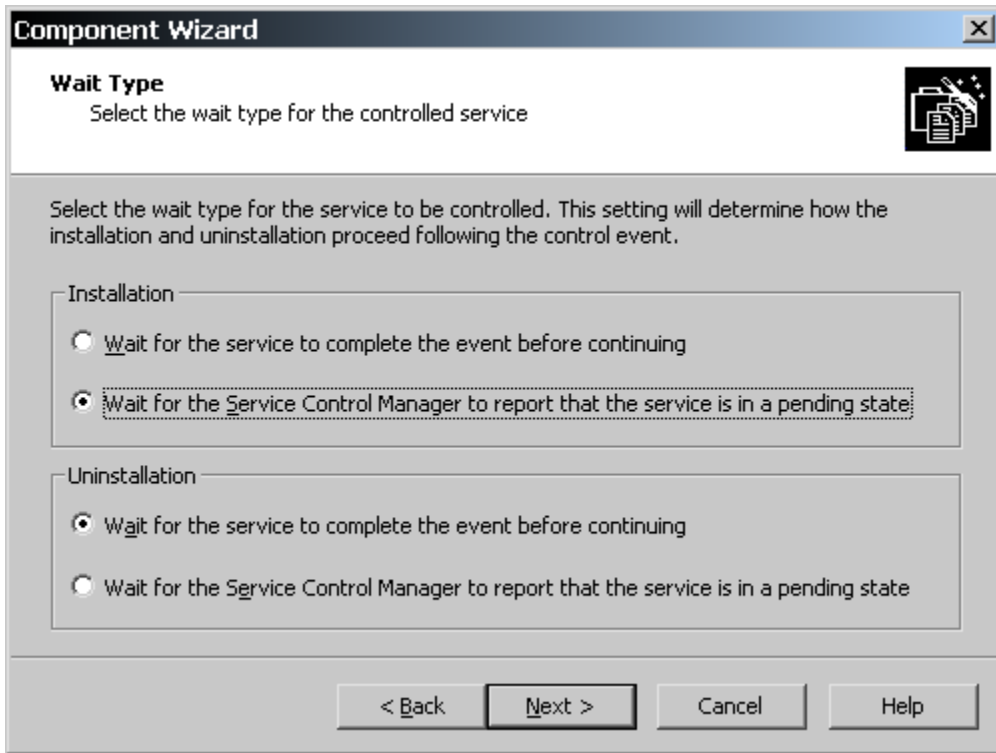


Figure 14-28: *Specifying the wait type for events to be performed during installation and uninstallation.*

You need to perform the same operations as just discussed for the second service. When you are finished, you should have what is shown in Figure 14-29.

When you have finished creating the second component for controlling the second service you will have three components under the SoundsService feature. For either of the components used to control the services, expand the tree under the Advanced Settings and click on the Control NT Services icon. Then click on one of the event names shown in the Control NT Services tree and you will see a property page that shown all the selections that were made when running the Component Wizard. You can edit these properties in this property page. The first event name (NewEvent1)

refers to the operation of installation and the second event name refers to the operation of uninstallation.

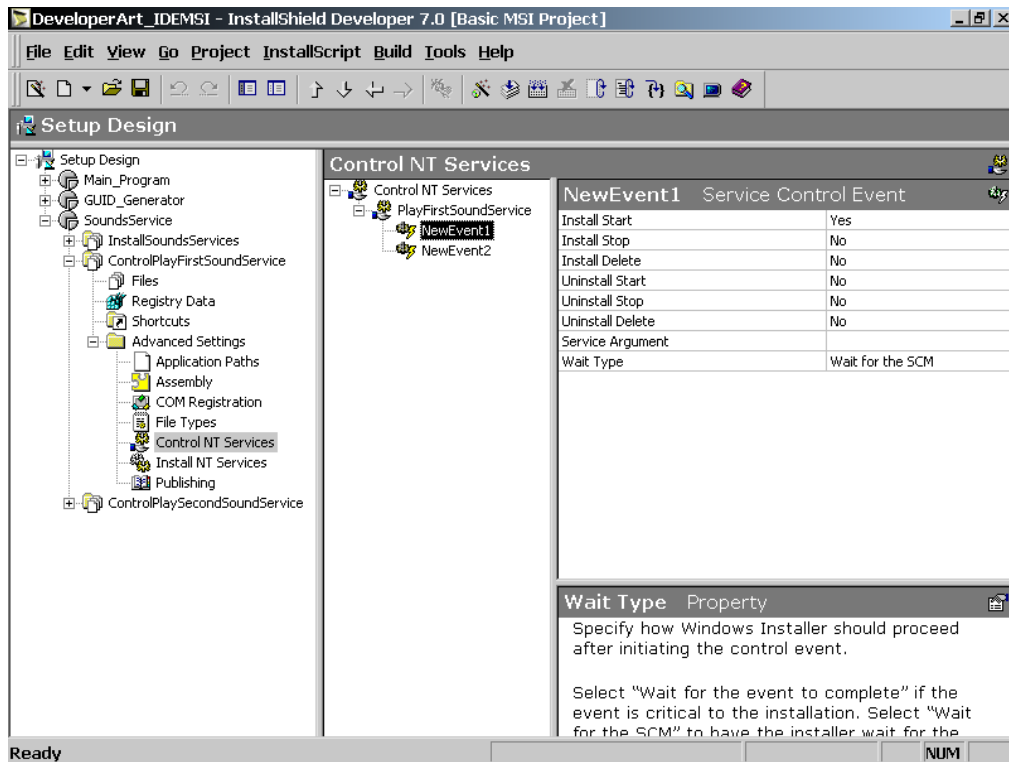


Figure 14-29: *The property page for the Control NT Service icon under Advanced Settings.*

You have used the Component Wizard to create three components to install and control the services contained in one executable. A better approach in this particular instance would have been to create the component that installs the two services and, within that component, enter the control logic under Advanced Settings. With this method, you have one component instead of three and it is easier to manage. After all, the components that are used for service control contain logic that is used only during the installation and uninstallation operations. These components have nothing to do with the functioning of the application after it is installed.

The same operations that you have carried out in a Basic MSI project can be performed in the same manner in a Standard project. The Standard project that is on the included CD_ROM has all the control logic placed in the same component that is

installing the NT services. To test this example, build the project and install the Developer Art application. You should hear the sounds generated by the NT services when they play the .wav files.

When the NT services are installed entries are made in the registry under the following key:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services
```

There will be two sub-keys under this key one for each of the services that you have installed. The value data written against these keys tell the Service Control Manager how to treat the services. The entries that are made for the PlayFirstSoundService service are shown in Figure 14-30.

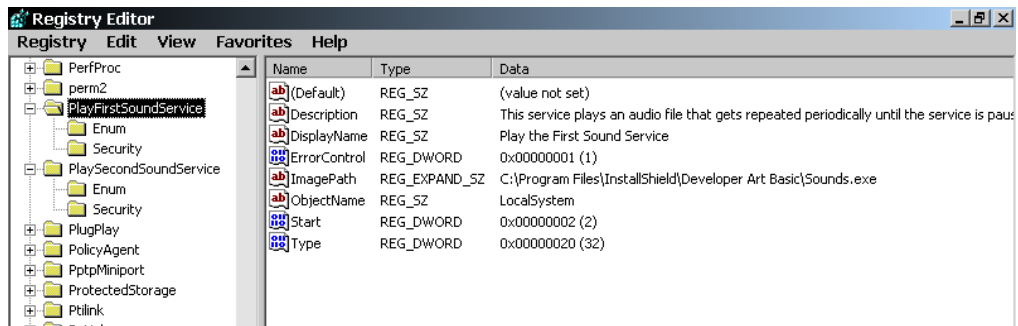


Figure 14-30: The data written to the registry for the PlayFirstSoundService service.

When you get tired of hearing the sounds that are played by the installed services, you can go to the Services Control Panel applet on Windows NT 4.0 or to the Services snap-in in the Microsoft Management Console in Windows 2000 and pause or stop the service. The Services snap-in is shown in figure 14-31.

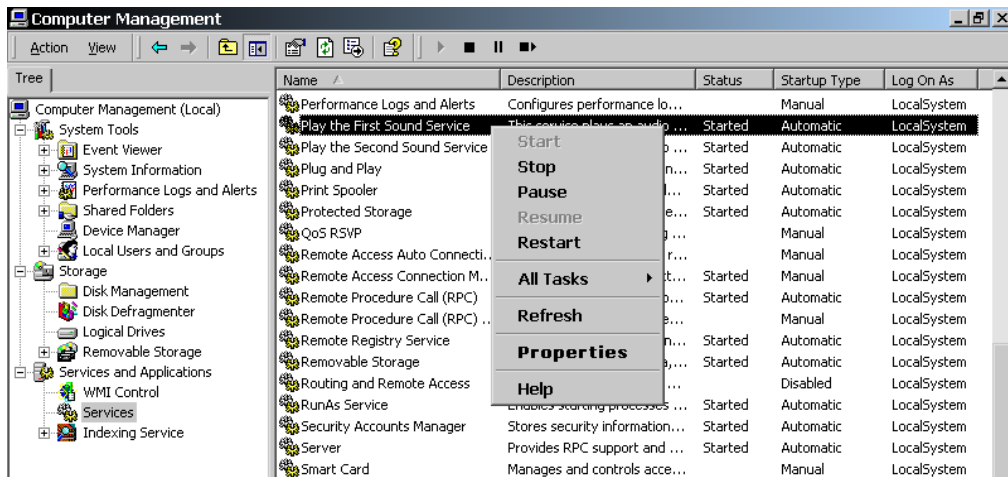


Figure 14-31: *The Windows 2000 Services snap-in showing the context menu for services.*

There are a few final subjects to discuss with regard to the installation of NT services. The first of these subjects is the interaction of an NT service with the desktop.

Interactive Services

This subject of a service interacting with the desktop is an interesting one. The first thing that you need to know about this subject is that a service should not be written so that it has to interact directly with the user. The whole purpose of having a client is so that the service can stay in the background and let the client provide the interaction with the user.

Understanding how a service interacts with the desktop brings into focus some little-known facts about how the Windows operating system works. There are two entities that are involved: the window station and the desktop. A window station is a secure object that contains a clipboard, a set of global atoms, and a group of desktop objects. The interactive window station assigned to the logon session of the interactive user also contains the keyboard, mouse, and display device. The interactive window station is visible to the user and can receive input from the user. All other window stations are non-interactive, which means that they cannot be made visible to the user, and cannot receive user input.

A desktop is a secure object contained within a window station. A desktop has a logical display surface and contains windows, menus, and hooks. A window station can have multiple desktops. Only the desktops of the interactive window station can be visible and receive user input. On the interactive window station, only one desktop at a time is active. This active desktop, also known as the input desktop, is the one that is currently visible to the user and that receives user input. There are typically three desktops that you see in any session on an NT based operating system. These are shown in the following list:

- The desktop that you see when you logon to the system or when you hit the Ctrl-Alt-Delete keys to see the NT task manager.
- The desktop that you see when running the shell.
- The desktop you see when a screen saver is running.

The system automatically creates the interactive window station. When an interactive user logs on, the system associates the interactive window station with the user's logon session. The system also creates the default input desktop for the interactive window station. When a non-interactive process such as a service application attempts to connect and no window station exists for the process' logon session, the system attempts to create a window station and desktop for the session. The name of the created window station is based on the logon session identifier, and the desktop is named "Default."

For a non-interactive service application to interact with the user, it must open the user's window station ("WinSta0") and desktop ("Default"). By default, only the logged-on user and service applications running in the LocalSystem account are granted access to the user's window station and desktop. This means that services running in other accounts must either impersonate the user when opening the interactive window station and desktop, or have access granted to those accounts by the user.

The above is the reason that for a service to interact with the desktop it needs to run in the local system account. Also, when a service shares a process with one or more other services, all services in the process must be allowed to interact with the desktop.

Installing an NT Service to a User Account

Most NT services are installed to run under the local system account. However, it is possible to install an NT service so that it runs only under a particular user's account. One of the main reasons that an NT service might be installed to run under a particular user's account is so that the service has access to network resources.

When you were creating the component that installs the example services, you saw in the Service Logon panel (Figure 14-21) a disabled “This account” option. The fact that this option was disabled is a holdover from the requirement that a service that shares a process with another service cannot be installed to a user account. With Windows 2000 and later, this is no longer a restriction and it is possible to install an NT service to a user account even if it does share a process with other services.

It does not matter that this radio button is disabled or enabled because you do not want to enter a name and password here even if the service is to be installed to a user account. If you enter a user name here and a password, the password will be placed in the Windows Installer database making it available to anyone who wants to open the database using Orca. The only real approach to installing a service to a user account is to permit the entry of the user name and password when the installation is run. This can be accomplished by using a custom dialog box where this information can be entered.

In Chapter 12 you created a custom dialog named `InstallNTService` that performs this function. For easy reference, this dialog is shown here in Figure 14-32.

When you created this dialog in the Basic MSI project, you were required to assign the name of a public property to both edit controls. The name of the property for the Domain\User Name field is `ACCOUNT` and the name of the property for the Password field is `PASSWORD`. When values are entered into these two fields, the properties take on the values that are entered. The trick then is to use these properties at run time to install the NT service to the user account that was entered.

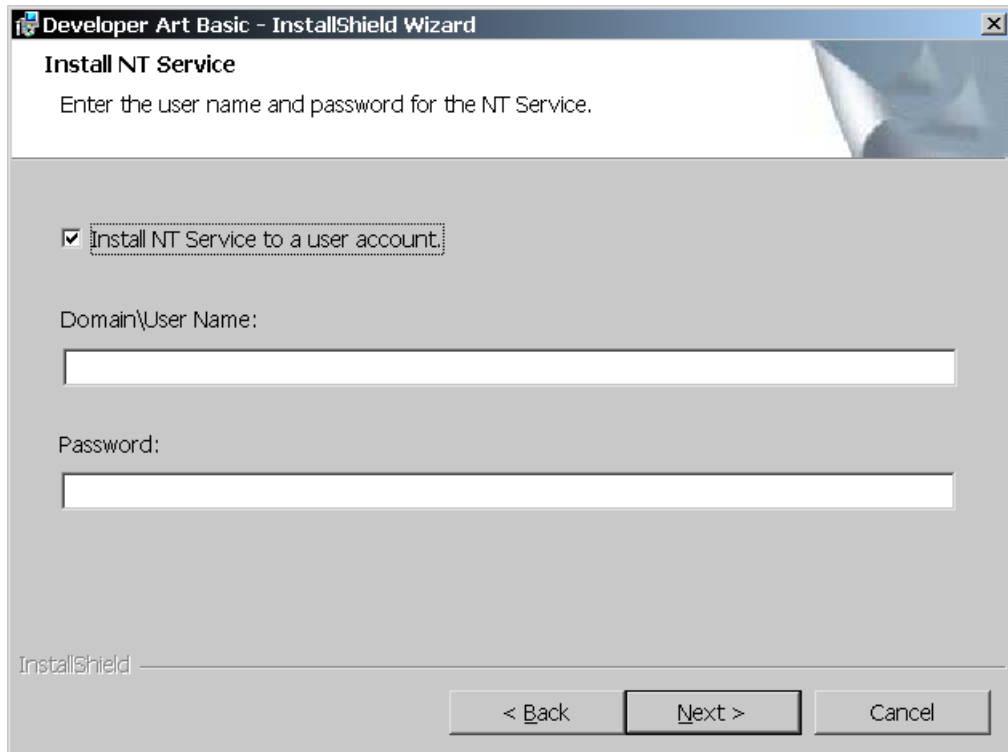


Figure 14-32: *The InstallNTService custom dialog created in Chapter 12.*

To make use of the property values that are entered at run time so that the NT service is installed to the user account, you need to add the name of these properties inside square brackets to the User Name and Password attributes for the service. This is shown in Figure 14-33, assuming that the PlayFirstSoundService service is to be installed to a user account. If the PlayFirstSoundService is not installed to a user account, the two strings [ACCOUNT] and [PASSWORD] are evaluated by the Windows Installer as NULL entries in these two fields.

To create the same functionality in a Standard project, you would still add the two property names inside square brackets as shown in Figure 14-33. You would have to add some code to your installation script in order to set the value of the ACCOUNT and PASSWORD properties.

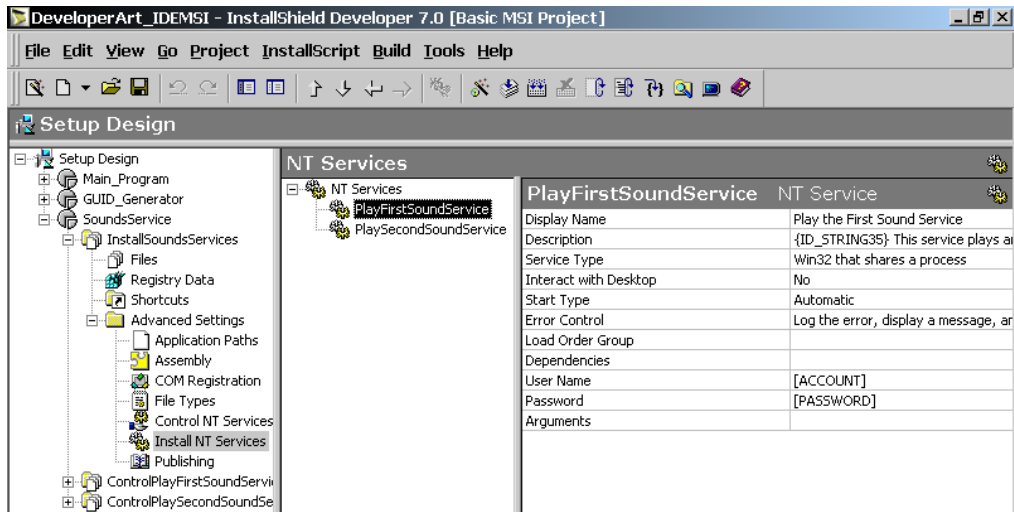


Figure 14-33: Setting up an NT service to be installed to a user account.

The revised code for the InstallNTService dialog that sets the values of the two properties is as follows:

```
Dlg_InstallNTService:
    szTitle = "";
    szMsg = "";
    nResult = InstallNTService(szTitle, szMsg, svUser, svPassword);
    if(nResult = BACK) goto Dlg_SdCustomerInformation;

    if(svUser != "" && svPassword != "") then
        MsiSetProperty(ISMSI_HANDLE, "ACCOUNT", svUser);
        MsiSetProperty(ISMSI_HANDLE, "PASSWORD", svPassword);
    endif;
```

All that is being done new here is to check if any values were entered in the InstallNTService dialog. If so, these values are used to set the values of the ACCOUNT and PASSWORD properties.

There are two important things that you need to know about installing an NT service to a user account. First, it still takes a person with administrative privileges on the local machine to perform the installation. Second, the user for whom the NT service is being installed needs to have the "Log on as service" user rights for the installation to be valid.

Font Components

The last type of component that can be created by the Component Wizard is a font component. Using the Component Wizard allows you to select font files that are already installed on the build machine or font files that are not installed on the build machine, but are available from some location. The purpose of the Component Wizard is to author the Font table correctly. The Font table has only two columns with the first column being a foreign key into the File table. The second column of the Font table is where the title of the font is placed, but only if there is no embedded title in the font file. True type fonts have an embedded font title, thus they should not have a value placed in the second column of the font table. Font files with a .fon extension are not true type fonts and do not have an embedded font title, so the font title used needs to be added explicitly to the Font table. For true type fonts, the Windows Installer reads the font title from the embedded name and makes the appropriate registry entry.

If a font component for a .fon font file is created from a font that is already installed, then the Component Wizard, using the title for the .fon file that is found in the registry, adds the font title automatically to the Font table. If a font component is created for a .fon font file that is not installed, then you have to specifically add the title that is to be used during the installation. Adding a title for a .fon font file can be accomplished in one of three different ways. It can be added through the Component Wizard, it can be added in the Properties dialog for the font file after the component is created, and it can be added by going to the Font table in the Direct Editor view.

By default, any font component created by the Component Wizard has a destination set as the fonts folder in the Windows directory. Fonts can be installed to other folders, but it is highly recommended to install fonts to the Fonts folder. When a font is installed, a registry entry under the following key is created.

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Fonts
```

The names written under this key are the font titles and the data values are the names of the font files.

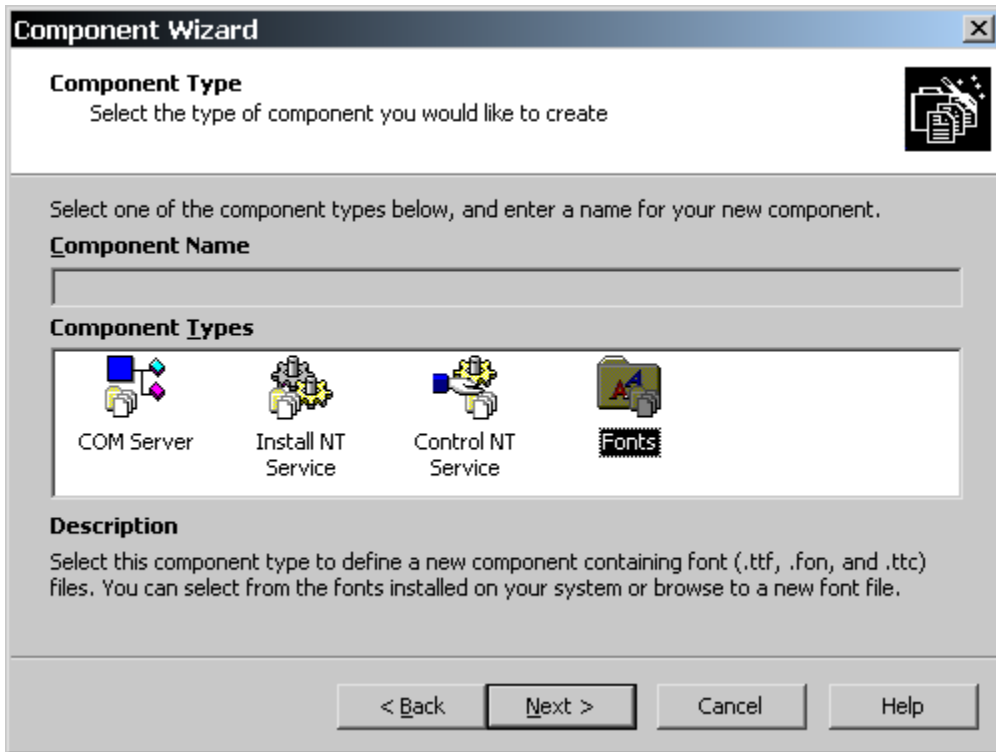


Figure 14-34: *The selection of the Fonts component type in the Component Wizard.*

There is no specific example for installing a font with the Developer Art application because it does not require any particular font files. However, it is instructive to look at the Component Wizard when the Fonts component type is selected (Figure 14-34).

Note that when the Fonts component type is selected, the Component Name field is disabled. This is because the Component Wizard creates a separate component for each font file that is selected. The Component Wizard provides default component names that use the name of the font file.

The next panel in the Component Wizard provides a selection of all the installed fonts to be found on the build machine. Below the list of installed fonts is an option that allows you to browse to other locations for fonts to include in the installation package. These other locations are where the font files that are not installed reside. If the option at the bottom of the Add Installed Fonts panel is selected, then the next

panel that is displayed in the Component Wizard is the Add New Fonts panel (Figure 14-35).

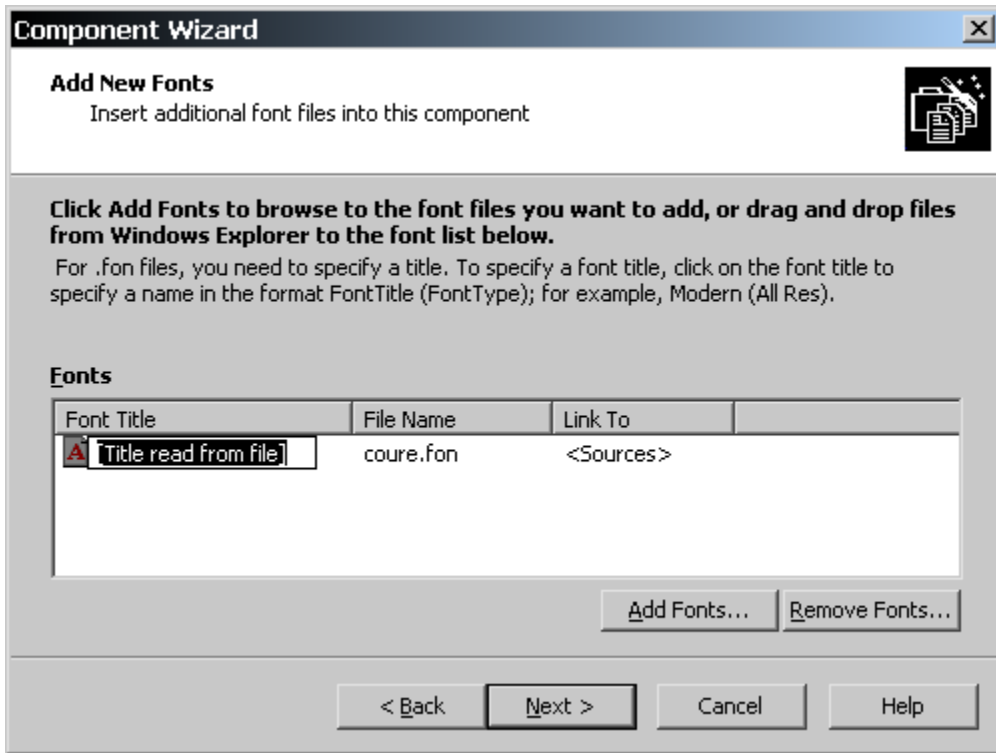


Figure 14-35: *Adding font components for fonts that are not installed on the build machine.*

Regardless of the type of font file that is added using the Add New Fonts panel, the Font Title column always displays the string [Title read from file]. However, if the extension used by the font file is .fon, you need to click once on this string and enter a font title that will be entered into the second column of the Font table. If you do not enter a title string for font files with a .fon extension, the Windows Installer displays an error during the installation stating that it cannot register the font file. A valid entry into the Font Title column of the Add New Fonts dialog is shown in Figure 14-36.

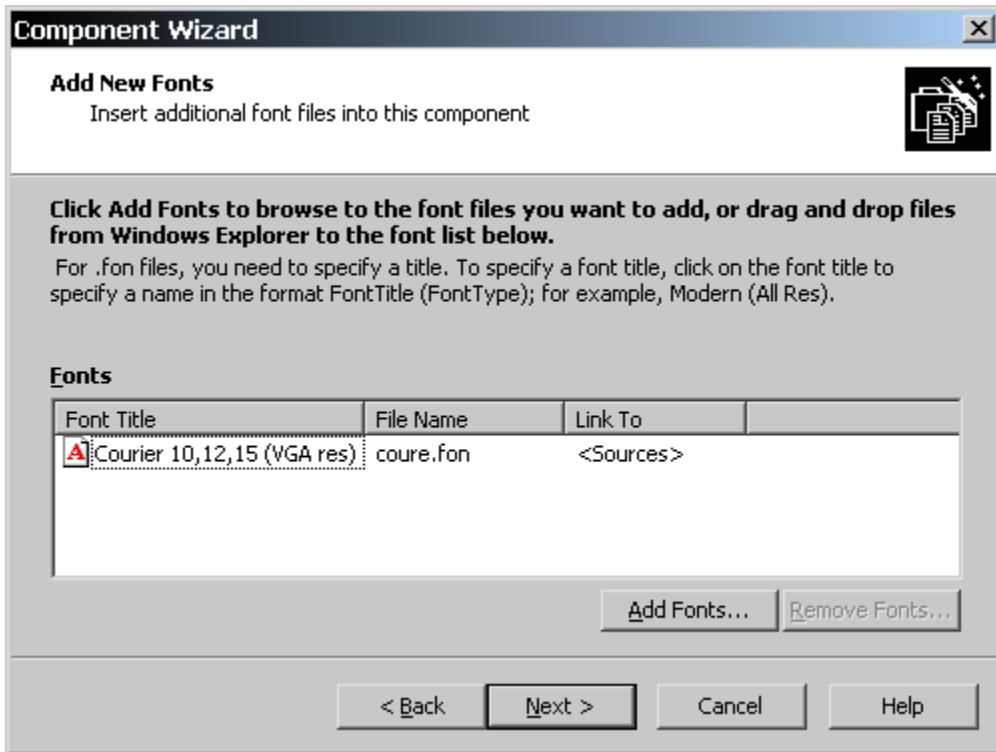


Figure 14-36: Adding a font title string for a .fon file not installed on the build machine.

The final panel of the Component Wizard provides a list of the components and font files that are being created.

One of the important things to remember is that a font component that is being installed to the Fonts folder should be made a permanent component. There is no reference counting mechanism for fonts except for the reference counting done for the component that installs them. This is doubly important on your build machine where you will be installing and uninstalling your application as you develop the installation package. If you have components that are installing fonts that are installed on the build machine, then you will find them missing the first time you test the install and then uninstall if the components have not been made permanent.

ODBC Components

Open Database Connectivity (ODBC) is an application programming interface (API) for database access. ODBC provides a single set of functions that can be used by developers to access data in various Data Base Management Systems (DBMS) that support SQL. To create components that install drivers, translators, and/or create data source names, it is important to understand how ODBC works. After the ODBC overview, you will add some more components to the Developer Art application that install a simple front end to a Microsoft Access database. This front end uses ODBC to access the tables in the Access database.

Overview of ODBC

The complete ODBC environment consists of five elements: the ODBC Administrator, the database application, the ODBC Driver Manager, the ODBC Driver, and the data source. The relationship of these five elements is shown in Figure 14-37. Each of these elements is discussed in the next sections.

ODBC Administrator

The ODBC Administrator is accessed through the Control Panel and is used to add, configure, and delete data sources from a system. It can also be used to view what drivers are installed on the local system. When you add or delete a data source from your system using the ODBC Administrator, it does not mean that you are actually installing a database file or uninstalling it from the system. Normally you use the ODBC Administrator to associate a name with a particular database. What this does is create entries in the registry. Depending on the ODBC drivers that are installed, you may also be able to create an empty database file that contains no tables. This is possible in the case of Microsoft Access.

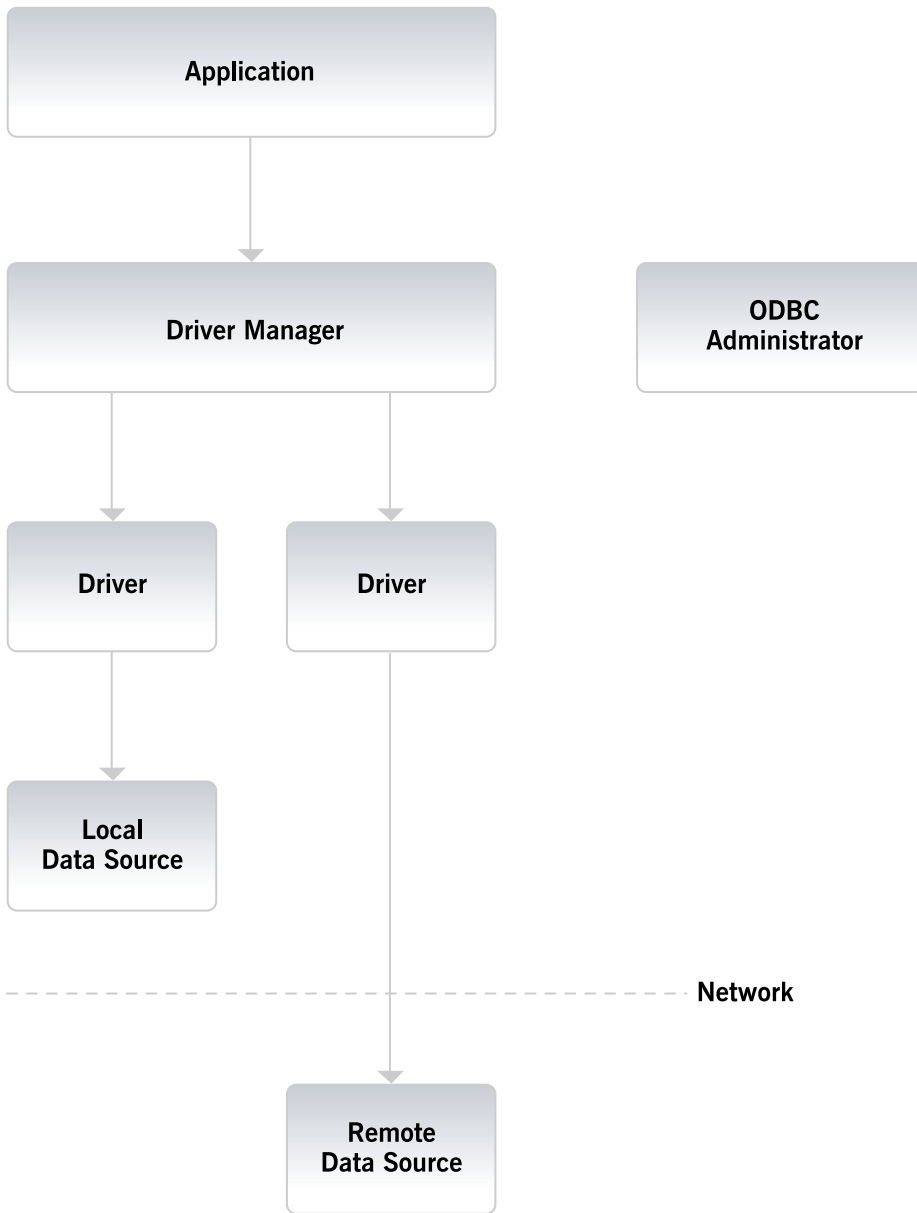


Figure 14-37: *The ODBC environment and the key players.*

The Application

The application element of the ODBC picture is the front end that is used to provide access to the database file in which the data is stored. It is in the application that the ODBC API function calls are made. Depending on how the application is written, it may offer the user a selection of databases with which they can connect. In the simple example that you will perform here, the application is tied to a particular database and it is the installation's responsibility to make sure that both the database file is installed and the proper registry entries are made so that the application can connect to the database.

The Driver Manager

The Driver Manager is responsible for managing the communication between the application and the specific ODBC driver that is used as the interface with the data provider. When an application tries to make a connection with a database, the Driver Manager loads the correct driver based on a search of the registry for the specified data source name. The Driver Manager is also responsible for unloading a driver that is no longer needed. The Driver Manager is considered the core of ODBC and it is installed using the Microsoft Data Access Components (MDAC). InstallShield Developer provides a merge module for installing MDAC.

The Drivers

Drivers are the elements of ODBC that process requests sent from the application via the Driver Manager. If necessary, a driver modifies a request from an application into a form that is understood by the data source. Drivers can make use of translation DLLs that provide a generic mechanism that allows the driver to translate data from one character set to another.

The Data Source

The final element of the ODBC environment is the actual data with which a connection is being made. A data source can be a file such as created by Microsoft Access or it can be a database on a server such as created by Microsoft SQL Server. Data sources can be local to everyone who uses the machine or they can be available only on a user-by-user basis. Data sources that are installed so that all users of the machine can have access are registered under the following registry key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\ODBC
```

It is necessary to have administrative or elevated privileges in order to install a data source for all users of the machine. These types of data sources are called system data sources.

Data sources that are installed only for a particular user are registered under the following registry key. This type of data source is called a user data source.

```
HKEY_CURRENT_USER\SOFTWARE\ODBC
```

Many ODBC drivers have setup DLLs that are used to properly install the driver on the target system. Translator DLLs also sometimes have a special setup DLL that is used to make the proper registry entries.

The focus of this section is how to create Windows Installer components that install ODBC drivers, ODBC translators, and ODBC data source names. A data source name is an entry in the registry that associates a name to an actual database that may be on the system somewhere.

The Windows Installer and ODBC

In the Windows Installer database schema, there are five tables devoted to the installation of ODBC drivers, translators, and data source names. Figure 14-38 shows the schema of these file tables and the two interfacing tables. The schema here uses the same format as described in Chapter 3.

There are also three standard actions placed in the `InstallExecuteSequence` table that have the task of reading the tables shown in Figure 14-39 and performing the actions that are required to install or remove the drivers, translators, and data source names identified in the ODBC related tables. These actions are `SetODBCFolders`, `RemoveODBC`, and `InstallODBC`. The only one of these actions that its purpose is not clear from the name is the `SetODBCFolders` action. The `SetODBCFolders` action checks to see if any of the drivers being installed on the target system already exist and if so it sets the installation folder for the new driver to be the same as for the existing driver.

functionality. ODBC drivers, translators, or data source names cannot be installed unless the Driver Manager is installed in advance.

The next section looks at how you can use InstallShield Developer to create the installation for a simple database application that uses ODBC to connect to an Access data source.

Creating and Installing ODBC Components

For this example, you will use the Basic MSI project created for the Developer Art application. You will need to create components for the Microsoft Access driver, the associated translator, and you will need to create a new data source name (DSN) that gets created under HKEY_LOCAL_MACHINE. You will then add another shortcut to the executable that accesses the data in the Access database.

The additional functionality that you will be adding to the Developer Art application in this example is a very limited capability to access a particular Microsoft Access database through an application that uses ODBC to make the connection. On the included CD-ROM are both the .mdb file and the front-end application that provides access to one of the tables in the database. The .mdb file contains only the tables because the forms and other data have been removed by splitting the original database file.

Open the Basic MSI Developer Art project and create a new top-level feature under which you can place all the new components that are related to the running of the database application. After you create the new top-level feature, create two new components by right clicking on the feature and selecting the New Component option. As shown in Figure 14-39 the names of these two components are DatabaseApp and Database.

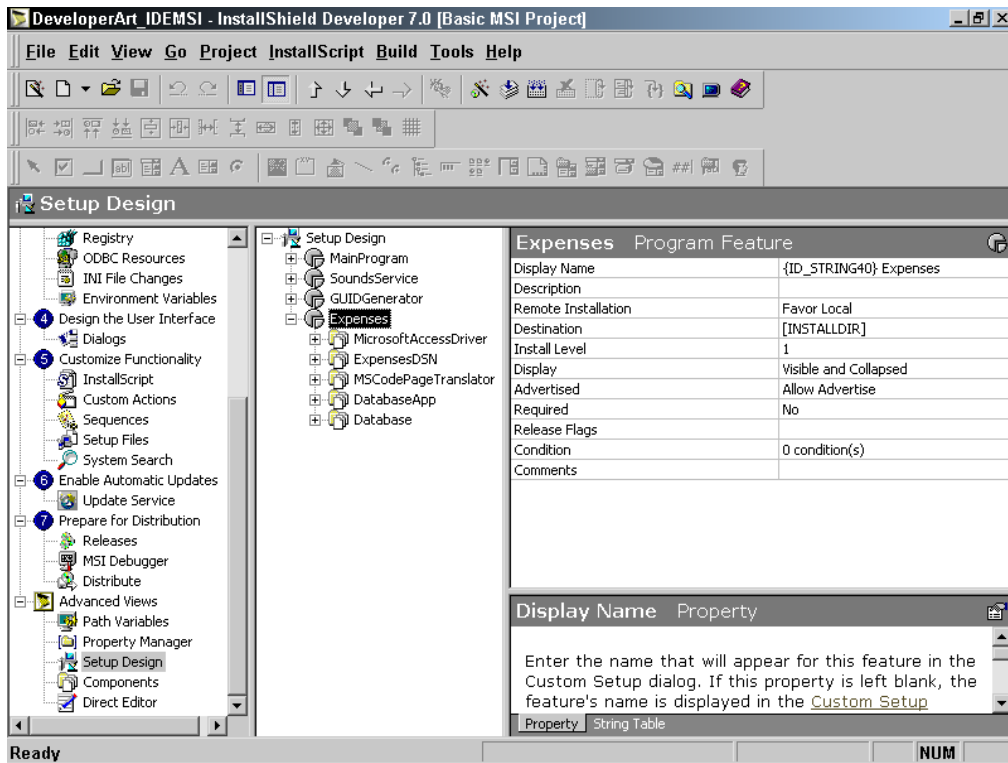


Figure 14-39: *The components required to install the database application.*

In the Files view for the DatabaseApp component, add the Expenses.exe file and make it the key path for the component. Also, for this component create a shortcut and add it to the folder where the other two shortcuts are. This is all you need to do for this component.

In the Files view for the Database component, add the Expenses.mdb file. You do not have to make this file the key path, but there is no problem if you do. Since this component is something that will most likely be updated by the user, set the Permanent property to Yes. This means that it cannot be removed when the application is uninstalled. Another thing that you want to do is to change the Destination property to a globally shared location and assign this location to an identifier named DATABASEDIR. The global location for applications that is used in the projects on the CD-ROM is as follows:

```
[CommonAppDataFolder]InstallShield\Shared Database
```

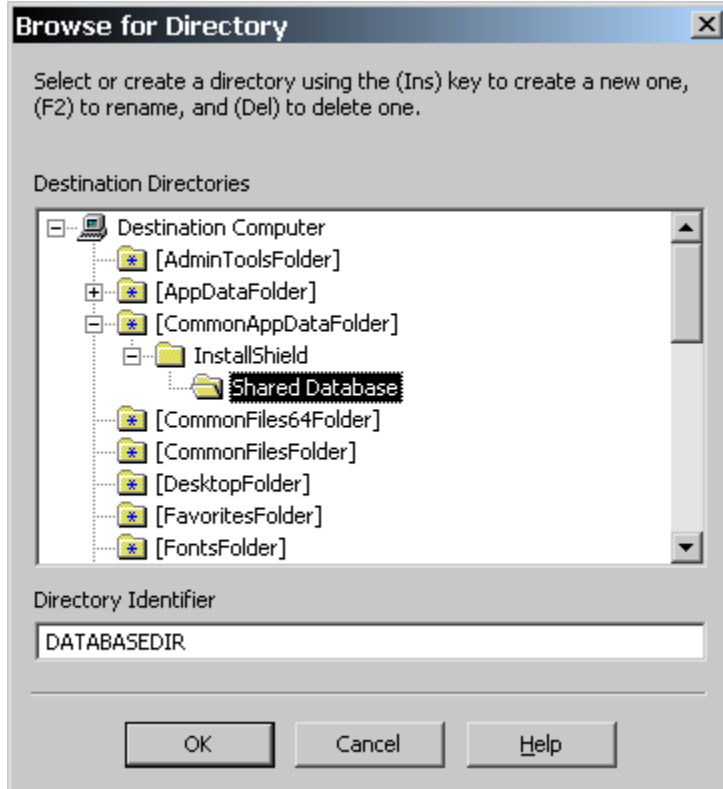


Figure 14-40: *Creating the DATABASEDIR directory identifier.*

This identifier is the one that is used in the DatabaseFolder dialog that is included in all default Basic MSI projects. In a Standard project the SdAskDestPath or AskDestPath dialogs would be used. To create this path identifier, select “Browse, create, or modify a directory entry...” from the drop-down menu in the Destination field. When you have created this directory identifier, the Browse for Directory dialog will look as shown in Figure 14-40. The identifier does not appear in the Destination field. Instead, the default location to which the directory identifier has been set appears in the Destination field.

By default, the DatabaseFolder dialog is not used in any of the user interface sequences so you have to add it to the InstallWelcome wizard sequence. You should place it after the InstallNTService dialog and before the SetupType dialog. As discussed in Chapter 12, you add a dialog into a wizard sequence by setting the targets

of the NewDialog control events for the Next and Back buttons in a Basic MSI project. For a Standard project, you need to add code to the OnFirstUIBefore event handler in order to use the SdAskDestPath or AskDestPath dialogs to browse for the location to install a database file.

You can now create the final three components for the driver, translator, and DSN. Do this by using the ODBC Resources view under Step3 in the View List. In the ODBC Resources view you will see all the drivers, translators, and data source names that are installed on the build machine. You cannot create components for drivers and translators that are not already installed on the build machine. For drivers you can modify the values of attributes and you can add new attributes. For translators you can only modify the value of the attributes but translators only have three attributes so you cannot add any new ones.

For this example, select the Microsoft Access Driver (*.mdb) driver entry in the ODBC Resources panel and the MS Code Page Translator entry. Right click on the Microsoft Access Driver (*.mdb) driver in the ODBC Resources panel and select the New DNS option. Name this new data source Expenses, as shown in Figure 14-41. With the Expenses DSN still selected, go to the Properties panel just below the ODBC Resources panel and name the DSN component ExpensesDSN.

You can accept the remaining default attribute values, but you need to create an additional attribute. The name of this attribute is DBQ and you want to give it a value equal to the location where the Expenses.mdb database file is going to be installed. Since you are allowing the end user to choose where to install this database file, enter the following value:

```
[DATABASEDIR]Expenses.mdb
```

You should take a look at the driver, translator, and data source components that were created. Note that all of these components have the Permanent property set to Yes. Also, the destination for each of these components has been set to the [SystemFolder] directory identifier. For each of these three components in the Setup Design view, you can go to the Advanced Settings tree to see that a new icon has been added there called ODBC Resources. The process of creating an ODBC component added this new icon. You can go to these icons and make changes to the attributes if you do not want to work in the ODBC Resources view under Step 3.

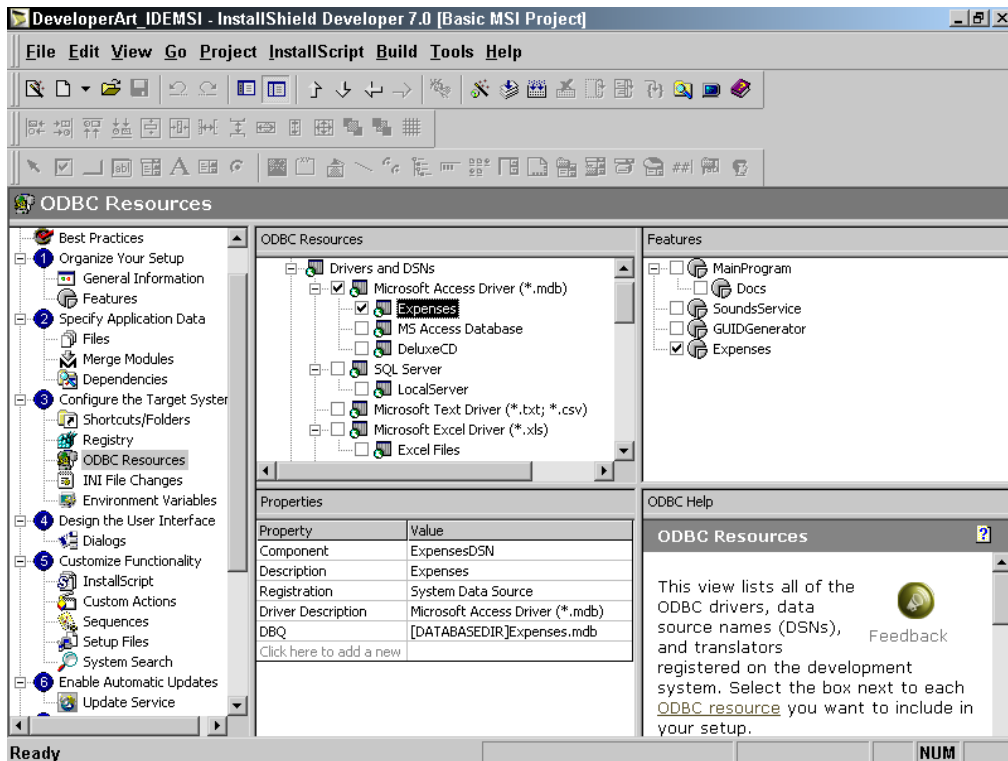


Figure 14-41: Creating the *Expenses* data source name.

Build the project and install this latest version of the Developer Art application. You will see that you now have a third shortcut and, when you click on it, a database editing form is launched that allows you to view and edit a table in the Expenses database.

Changes Made to the Operating System for ODBC

To the subject of ODBC, we should look at the changes that are made to the target system when ODBC components are installed. First, you need to look at the following registry key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\ODBC
```

Under this key there are two sub-keys as shown in Figure 14-42. The ODBC.INI key is where the data source names are registered for all users of the machine.

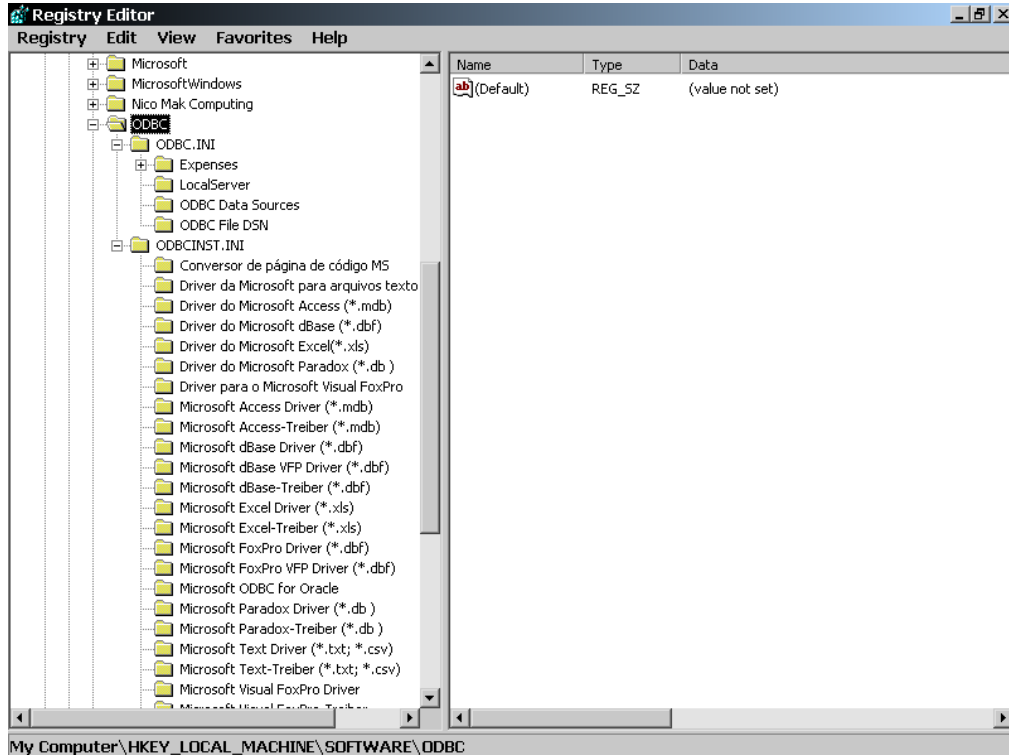


Figure 14-42: *The ODBC registry keys under HKEY_LOCAL_MACHINE.*

There is a similar key under the following key where data source names for individual users are created:

```
HKEY_CURRENT_USER\Software\ODBC
```

The second sub-key is named ODBCINST.INI and this is where all the drivers are registered. Drivers are only registered under HKEY_LOCAL_MACHINE.

There is a final point that needs to be made about the changes to the system when ODBC components are installed. Most of the ODBC drivers that are created and shipped by Microsoft are part of the Windows File Protection (WFP) list. The WFP

is discussed at the end of Chapter 13 but it is a mechanism on Windows 2000, Windows ME, and Windows XP where critical system files cannot be replaced by installation programs. Testing this particular installation on Windows 2000 will make all the registry entries but it will not copy either the driver or the translator files to the System32 folder.

We can now move on and see how to create components in a way so that they can easily be shared between applications.

Merge Modules

Merge modules provide a standard method by which setup developers can deliver shared Windows Installer components and setup logic to multiple applications. Merge modules are used to deliver shared code, files, resources, registry entries, and setup logic to applications as a single compound file.

A merge module is similar in structure to a simplified Windows Installer .msi file except that it has an .msm extension. However, a merge module cannot be installed alone. It must be merged into an installation package using a merge tool. InstallShield Developer provides the functionality to both create and use merge modules.

When a merge module is merged into an .msi file for an application, all the information and resources required to install the component contained in the merge module are incorporated into the application's .msi file. The merge module is then no longer required to install the components and the merge module does not need to be shipped as part of the media image. Because all the information needed to install components is delivered as a single file, the use of merge modules can eliminate many instances of version conflicts, missing registry entries, and improperly installed files.

In the next section, we will work through a simple example of creating and using a merge module in the Developer Art application. Using the Basic MSI project for the Developer Art application, you will create a merge module for the ArtWork.dll file and then use the merge module in place of the component that was created for this file in Chapter 5. There is no difference in how merge modules are created and used in a Basic MSI project and in a Standard project.

Creating a Merge Module

Merge modules are created using the Merge Module Project icon in the “Create a new project...” sub-view under the InstallShield Today view. Merge module projects have the same extension (.ism) as used for both the Basic MSI and Standard projects. You can also create a new merge module project by using the New option on the File drop-down menu.

For this example, use either of the approaches mentioned above to create a merge module project called ARTWORK.ISM.

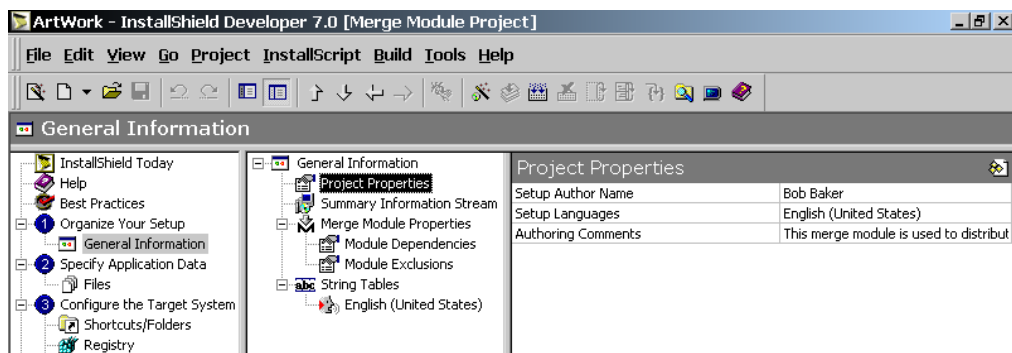


Figure 14-43: *The General Information sub-views in a merge module project.*

When you create the merge module project, you will see an IDE that is fairly similar to the one in which you have been working since Chapter 5. The main differences you will notice are that there is no Features view or Setup Design view. These two views are missing because you do not define features in a merge module, only components. The Merge Modules, Dependencies, Setup Files, and InstallScript views are also missing and there is no view that is concerned with the sequences of actions.

You start off with this merge module project by making entries in the sub-views found under the General Information view just the same as you did back in Chapter 5. The sub-views in a merge module project, however, are different than they are in a main installation project, as shown in Figure 14-43.

The sub-views under the General Information view are described along with a discussion of the various fields in which you need to make entries.

Project Properties: The fields for this sub-view are exactly the same as for a main installation project. You can enter any values that you want here. Leave the default language selection.

Summary Information Stream: The fields for this sub-view take the same information as for a main installation project with one exception and that is the Subject property. The Subject property provides a name for the merge module, but this name is not used in the main installation project. The string that is entered for the value of the Subject property is used to identify the merge module in the list of merge modules that is viewable in a main installation project. A good entry for the merge module you are creating is "ArtWork Module for Creating Geometric Shapes". You will see this string again when you include the merge module in the main installation project for the Developer Art application.

Merge Module Properties: This sub-view contains a set of properties that you have not seen before. The Product Name property is used to identify the name of the merge module file that will be created. A good name for this property is ArtWork as shown in Figure 14-44. This name cannot contain any spaces because this value is used to create part of the primary key in the ModuleSignature table in the merge module database. The Product Version property is the version of the merge module and is used to populate the ModuleSignature table in the merge module database. For this example, you do not have to change the value of this or any of the other properties shown in Figure 14-44.

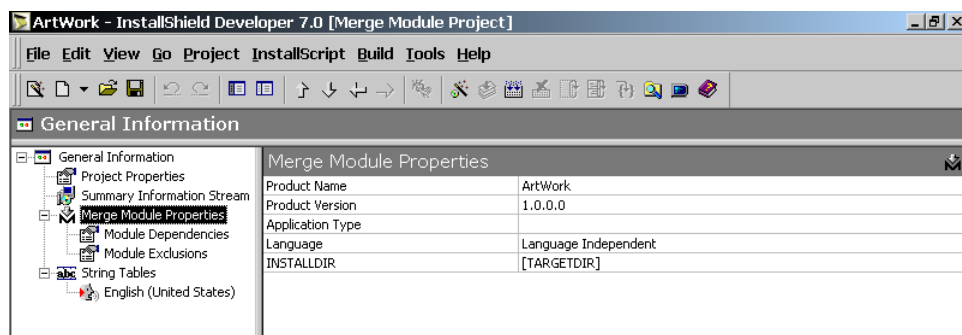


Figure 14-44: *The Merge Module Properties sub-view for the ArtWork merge module.*

Module Dependencies and Module Exclusions: In the Module Dependencies sub-view, you can identify other merge modules that are required

for the merge module being created to work properly. In the Module Exclusions sub-view, you can do the opposite. Here you identify any merge modules with which the current merge module cannot work. For this example, you do not have to do anything with either of these two sub-views.

String Tables: In a merge module, by default, there are no predefined string IDs and strings. This is because by default merge modules do not have a user interface, though this is possible. Only special merge modules have a user interface and then only because the merge module is distributing components that need special setup when they are installed. For this example, you do not need to do anything in the String Tables sub-view.

After filling in the General Information for the merge module project as just described there is only one additional activity that needs to be performed. That is to create the component that is to be delivered by the merge module. Do this using the Components view under Advanced Views. Go to the components view, right-click on the Components icon in the middle tree and select the Component Wizard option, as shown in Figure 14-45.

From here on, the creation of the component is exactly the same as we have already discussed except here you can let the Component Wizard extract the COM information for you instead of entering it in manually.

After you have created the component, build the merge module. Click the Build button on the toolbar in the merge module project to create the default build. Part of the default build process is to copy the merge module after it has been created to a central location where it can be accessed by any project that needs it. This location is as follows:

```
%USERPROFILE%\My Documents\MySetups\MergeModules
```

When the merge module is placed in this location, it appears in the merge module view of the main installation project where it can be selected.

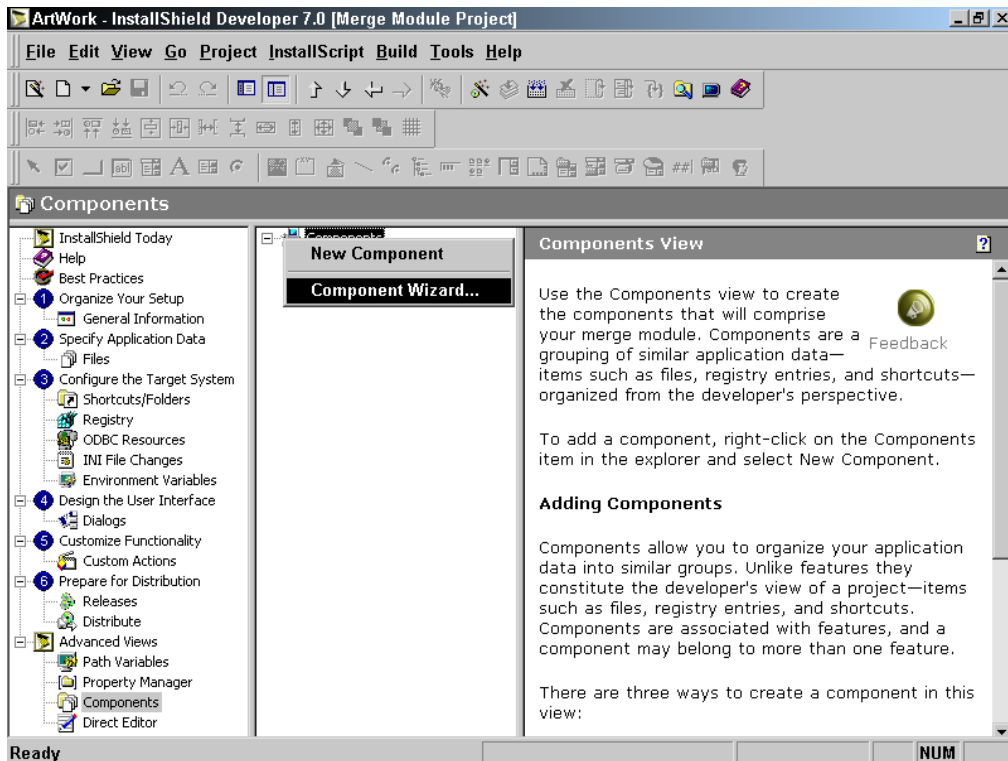


Figure 14-45: Accessing the Component Wizard in a merge module project.

Using a Merge Module

To use your merge module in a project, do the following:

1. Open your main installation project for the Developer Art application.
2. Delete the original ArtWork component from the project.
3. Go to the Merge Modules view (Figure 14-46). Find the name of the merge module as entered in the Product Name property of the Summary Information Stream sub-view in the merge module project. Select this merge module and make sure that in the upper-right panel, the MainProgram feature is selected.

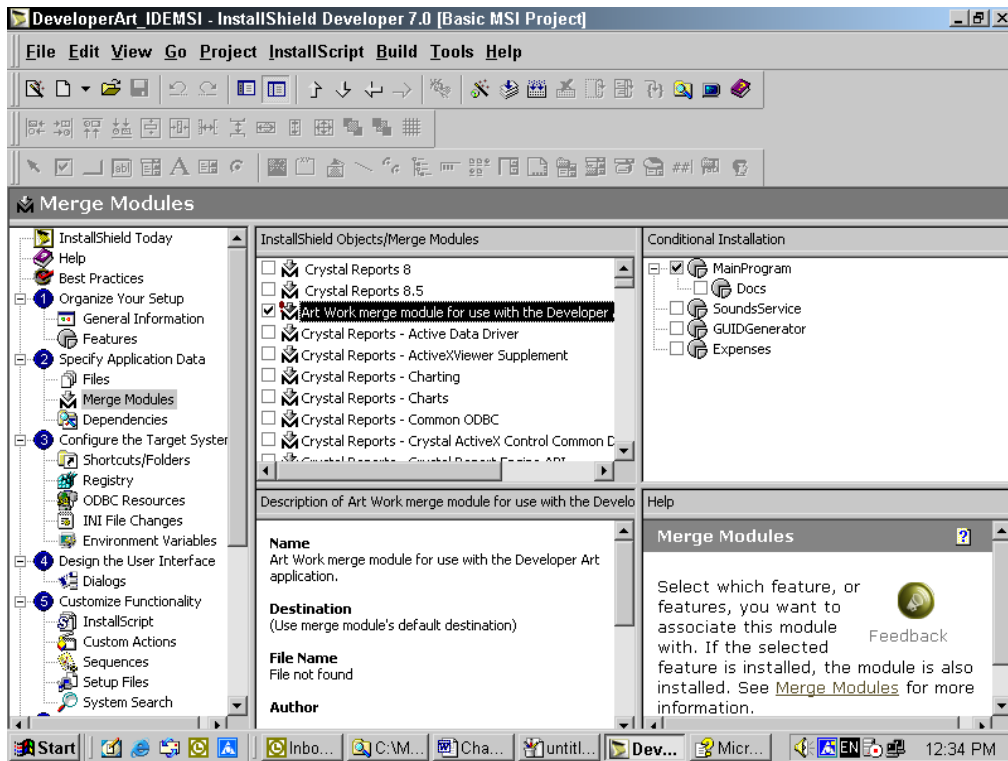


Figure 14-46: *The Merge Modules view showing the newly created merge module.*

Now there is one last operation to perform before you can build your main project. You need to make sure that the file in the ArtWork merge module goes to the same location as all the other files in the Developer Art application. If everything were left as it is at this point, the ArtWork.dll file would be installed to the root of any drive that had the largest amount of free space.

To make the necessary modification to the merge module, right-click on the name of the merge module in the Merge Modules view and select the Properties option. The Merge Modules Properties dialog is displayed (Figure 14-47). In the Destination drop-down menu, select the [INSTALLDIR] entry.

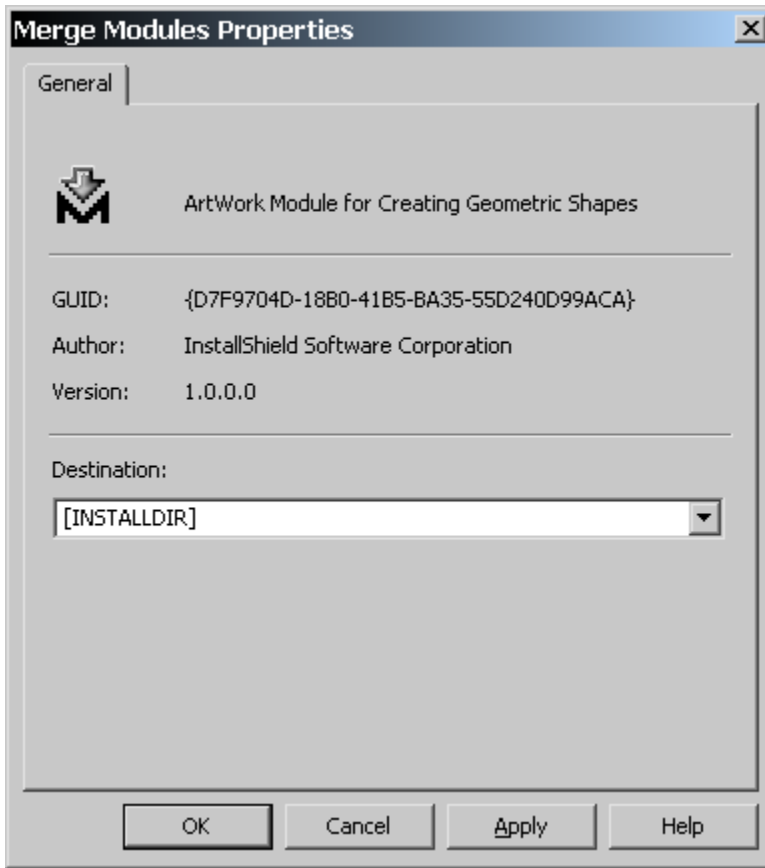


Figure 14-47: *The Merge Module Properties dialog.*

Build the main project and test the installation. When you test the application after the installation, you will see that it works just the same as in always did.

The general rule for creating merge modules is that you want to place your components into merge modules if they are going to be shared between applications of the same company or between applications of more than one company. If the component is one of a kind and will never be shared between applications, then the component should be created in the main installation project and not placed in a merge module. You need to remember that COM components have global properties and any COM server that is to be shared across applications needs to be placed into a global location. This is because the location of the file that implements the COM server is part of the information that is placed in the registry.

In general, the rule is that if a component is to be shared across applications of the same company, then the shared location for these components would be under the Common Files folder: An example of a company wide shared location is as follows:

```
C:\Program Files\Common Files\InstallShield\Shared
```

If a component is to be shared across applications of more than one company, then the install folder for these components should be the System32 folder. This location is as follows:

```
C:\WINNT\system32
```

Keep in mind the discussion about how components are reference counted by the Windows Installer, as well as the rules for creating components where two different components should never share any resources, which includes the name, and install location for files. These subjects are covered in Chapter 13.

Conclusion

In this chapter you extended your knowledge of components by getting into the details of how to create special types of components. In the discussion of COM components you saw the difference between a standard DLL and a COM DLL. You also worked through the creation of components that install an executable that contains two NT services. Many of the details of how NT services work are covered as part of the discussion of how to create installations for them. The final type of component that can be created with the Component Wizard was the Font component.

Finally at the end of the chapter you worked through a live example of installing an ODBC application where the correct creation of the data source name was necessary to make the application work. The chapter ended with a very brief discussion on creating merge modules in which components can be shared between many different applications.

The main point that was stressed in both Chapters 13 and 14 is that the proper creation of components is important to getting an installation to work correctly under all circumstances.

Appendices



The CD-ROM

The main content of the CD-ROM at the back of the book is the example projects and InstallScript code explained in the book. The example applications that are used to demonstrate various installation creation techniques were created with Microsoft Visual C++ version 6.0 and Visual Basic version 6.0. You can copy these example projects and source code directly from the CD-ROM but then you will have to remember to remove the Read-only attribute. There is an installation program that allows you to install all the examples and source code to a destination of your own choosing.

All the InstallScript source code that can be compiled and run is contained on the CD-ROM using the figure number as the name of the .rul file. You can take this code and copy and paste it into the InstallShield Developer Script Editor and run the example after first compiling.

Also included on the CD-ROM is an evaluation copy of version 7.03 of InstallShield Developer. This is a time locked copy that will stop working after a period of time. This is to just get you going but you should obtain a purchased copy of the product

A P P E N D I C E S

for any long-term work with the material in the book. To be able to install this evaluation copy you will have to go to the following Web site and enter your contact information. After entering your contact information a password will be e-mailed to you and you will use this password to unlock the evaluation copy.

`http://www.installshield.com/ispress`

There is also one or two white papers included on the CD-ROM that have to do with subject matter that is not covered in the book.

Index

A

- abort* statement, 377
- ActionText table, 130
- Active Data Object, 549
- ADMIN top-level action, 125, 138, 139, 155
- AdminExecuteSequence table, 125
- Administrative install, 15, 197
- AdminUISequence table, 125
- ADVERTISE top-level action, 125, 138, 139
- Advertisement, 14–15, 198–201
 - Basic MSI project, 200–201
 - Standard project, 198–200
- AdvtExecuteSequence table, 125, 138
- AdvtUISequence table, 125, 138
- Application Design tables
 - Component table, 115–17
 - definition, 106
 - Directory table, 118–19
 - Feature table, 112–15
 - FeatureComponents table, 117–18
 - schema, 109–19
- AppSearch table
 - schema, 622
- Array data type, 324–26

B

- Basic MSI project, 88
 - custom dialogs, 774–86
 - Dialogs view, 721–23
 - serial number input, 786–92
 - subscription in user interface, 792–95
 - user interface, 774–95
- Best Practices view, 37
- Best Practices Wizard, 248–51
- Binary table, 131

C

- Cabinet files, 96–98
- cdecl keyword, 434, 460
- Certified for Windows logo, 51

- file association requirements, 554–55
- Child installs, 17
- COM. *See* Component Object Model
- COM Structured Storage, 90–94
- Companion files, 848–49
- Compliance checking
 - using tables, 640
- CompLocator table
 - schema, 624
- Component Object Model, 12
 - compared to Win32 DLLs, 852–62
 - components, 852–74
- Component table, 115–17, 650
- Component Wizard
 - Best Practices operation, 825–30
 - COM components, 866–74
 - control NT service components, 893–99
 - font components, 906–9
 - install NT service components, 882–93
 - NT service components, 882–900
- Components
 - COM, 852–74
 - COM vs Win32, 852–62
 - companion files, 848–49
 - Component Wizard and COM, 866–74
 - creating ODBC components, 915–19
 - creation rules, 111–12, 806–11
 - defining registry entries, 581–84, 567–87
 - distribution, 159–60
 - dynamic file linking, 812–25
 - environment variables, 587–604
 - File Types, 555–64
 - font, 906–9
 - initialization files, 605–18
 - interfacing with legacy applications, 840–43
 - isolated, 845–46
 - key path, 61
 - make up, 803–6
 - NT service control, 893–99
 - NT service install, 882–93
 - NT services, 874–905
 - ODBC, 910–21

INDEX

- qualified, 847–48
- reference counting, 800–803
- removing registry entries, 584–87
- scanning, 830–40
- self-registration of COM, 862–66
- sharing COM components, 864–66
- transitive, 846–47
- Windows File Protection, 843–45
- Components view, 270–78
- Compressed GUID, 799
- Control table, 131
- ControlCondition table, 131
- ControlEvents table, 131
- Create a New Project view, 33–35
- CreateFolder table, 650
- Custom Action Wizard, 676
- Custom actions, 125, 155, 157, 154–59, 162
 - accessing database tables, 688–93
 - accessing the Binary table, 693–97
 - categories, 155–57
 - creating InstallScript, 661–78
 - creating without Custom Action Wizard, 679–80
 - Custom Action Wizard, 667–76
 - getting property values, 682–83
 - InstallScript function prototype, 400–401
 - InstallScript run-time architecture, 211–15
 - placing in sequence table, 676–77
 - procedure for working with database tables, 686–88
 - setting property values, 684–85
 - types, 157–59
 - using MsiDoAction function, 700–702
- Custom Actions view, 268–69
- Custom dialogs
 - basic dialog template for Standard project, 741–52
 - Basic MSI project, 774–86
 - dialog function for basic dialog template, 752–57
 - silent install in Standard project, 770–73
 - Standard projects, 738–73
 - using dialog templates in Standard project, 761–70
- Custom setup dialog, 71

D

- Darwin, 83, 134
- Darwin Descriptor, 799, 800

- Database
 - relational model, 96
 - SQL, 96
 - validation, 137
- DATABASE property, 143, 145, 151
- Database schema diagrams, 107–8
- Database tables
 - ActionText table, 130–31
 - Application Design tables - schema, 109–19
 - Binary table, 131
 - Component table, 115–17
 - Control table, 131
 - ControlCondition table, 131
 - ControlEvents table, 131
 - Dialog table, 131
 - Directory table, 118–19
 - Error table, 132
 - EventMapping table, 132
 - Feature table, 112–15
 - FeatureComponents table, 117–18
 - File table, 121–22
 - Icon table, 135–36
 - Media table, 122
 - RadioButton table, 132
 - Shortcut table, 134–35
 - table categories, 105–7
 - TextStyle table, 132
 - UIText table, 132
- Desktop Integration tables
 - definition, 106
 - Icon table, 135–36
 - schema, 133–36
 - Shortcut table, 134–35
- Developer Art sample application
 - description, 46–47, 220
- Dialog table, 131
- Dialogs
 - basics, 704–13
 - compiling resource files, 719–21
 - controls, 706–9
 - defining, 704–6
 - functions, 709–13
- Dialogs view
 - Basic MSI project, 721–23
 - compiling resource files, 719–21
 - Standard project, 714–19
- Dictionary object, 523–32
- Direct Editor, 650
 - RemoveRegistry table, 584–87
- Directory identifier

- curly braces, 52
- Directory table, 118–19, 650
 - resolving, 146–50
- DLL functions
 - calling a function in a user-defined DLL, 434–41
 - calling a function in a Windows DLL, 441–45
 - passing arrays to DLL functions, 445–48
- DLL redirection, 845–46
- DrLocator table
 - schema, 627–28
- Dynamic file linking, 812–25
 - creating, 812–16
 - options, 816–25

E

- Empty folders
 - creating, 650–52
- Environment table
 - schema, 589–93
- Environment variables, 587–604
 - accessing during install, 601–4
 - overview, 588–89
 - per-machine, 598–600
 - per-user, 598–600
- Environment Variables view, 593–97
- Error table, 132
- EventMapping table, 132
- Exception handling
 - Err object, 453–55
 - hierarchy, 455–59
 - InstallScript engine exceptions, 459–62
 - try* statement, 452–53
- exit* statement, 377

F

- Feature, 8
- Feature table, 112–15
- FeatureComponents table, 117–18
- Features view, 232–42
- File associations, 554–67
 - Certified for Windows logo requirements, 554–55
 - File Types for components, 555–64
 - MIME types, 564–67
- File Copy tables
 - definition, 106
 - File table, 121–22

- Media table, 122
 - schema, 119–22
- File menu, 38
- File table, 121–22
- Files
 - compressed, 96–98
 - copying, 11–12
 - costing, 9–10
 - properties, 62–63
 - Vital property, 63
- Files view, 244–48
- FileSystemObject object, 484–523
- Fonts, 906–9
- for* statement, 368–70
- Fresh install, 14
- Fresh install run-time architecture, 164–89
 - Basic MSI project, 184–89
 - Basic MSI project with InstallScript custom actions, 184–87
 - Basic MSI project without InstallScript custom actions, 187–88
 - program block execution, 178–79
 - Standard project, 164–84
 - uninstall log, 179–81

G

- General Information view, 221–32
 - Add/Remove Programs, 225–28
 - Product Properties, 228–31
 - Project Properties, 221–24
 - String Tables, 231–32
 - Summary Information Stream, 224–25
- goto* statement, 376

H

- Help view, 37
- Hungarian Notation, 303, 304

I

- Icon table, 135–36
- IDE. *See* Integrated Development Environment. *See* Integrated Development Environment
- if* statement, 361–65
- INI File Changes view, 609–14
- IniFile table
 - schema, 606–8
- IniLocator table

INDEX

- schema, 626–27
- Initialization files, 605–18
 - accessing during install, 615–18
 - creating and modifying, 609–14
- Install location, 51
- INSTALL top-level action, 125, 138, 139, 141, 142, 150, 155
- Installation Procedure tables
 - definition, 106
 - schema, 124–28
- Installation Validation table
 - definition, 106
 - schema, 136–38
- INSTALLDIR property, 51, 57, 58, 65, 71, 114, 147, 149, 150, 152, 231, 235, 236, 237, 244, 245, 246, 247, 248, 278, 582, 596, 597, 609, 613, 726, 727, 735, 736, 760, 820, 821, 867, 926
- InstallExecuteSequence table, 125, 143, 144, 151, 152, 153
- InstallScript
 - accessing database tables, 688–93
 - accessing the Binary table, 693–97
 - adding functions to the Function Wizard, 428–31
 - arithmetic expressions, 338
 - arrays of structures, 431–34
 - bitwise expressions, 351–58
 - built-in data types, 306–24
 - built-in function categories, 383–86
 - calling functions in a user-defined DLL function, 434–41
 - calling functions in a Windows DLL, 441–45
 - creating custom action, 661–78
 - creating custom actions programmatically, 700–702
 - custom action target function, 662–67
 - Custom Action Wizard, 667–76
 - dialog function, 752–57
 - Dictionary object, 523–32
 - engine exceptions, 459–62
 - Err object, 453–55
 - event handler function categories, 389–90
 - event handler functions, 390–99
 - exception handling, 452–62
 - exception handling hierarchy, 455–59
 - execution model, 21–23
 - FileSystemObject object, 484–523
 - function basics, 380–81
 - function prototypes - built in functions, 381–83
 - Function Wizard, 386–88
 - getting property values, 682–83
 - iteration statements, 368–75
 - jump statements, 376–77
 - OnBegin and OnEnd event handlers, 698–700
 - passing an array to a DLL function, 445–48
 - passing arguments by reference, 405–7
 - passing strings to functions, 448–49
 - prototyping custom action functions, 400–401
 - prototyping generic functions, 401–2
 - recursion, 404–5
 - relational and logical expressions, 339–46
 - script libraries, 408–28
 - selection statements, 361–68
 - setting property values, 684–85
 - SizeOf and Resize operators, 359–61
 - string expressions, 347–51
 - user defined data types, 324–30
 - variable naming, 302–6
 - Windows Installer automation interface, 464–83
 - WSH objects, 533–49
- InstallScript run-time architecture, 207–15
 - custom actions, 211–15
 - installing the InstallScript engine, 207–8
 - program block and event handlers, 208–11
- InstallShield Developer
 - Best Practices view, 37
 - Help view, 37
 - InstallShield Today page, 37
 - introduction, 26–28
 - New Project view, 33–35
 - Open a Project view, 35–37
 - opening page, 30–31
- InstallShield Professional, 20–21
- InstallShield Software Corporation, 3
- InstallShield3, 19
- InstallUISequence table, 125, 126, 143, 151, 152, 153
- Integrated Development Environment, 20, 30
 - Basic MSI project, 293
 - Standard project, 218–93
- Internet Explorer object, 549

K

Key path, 61, 64

L

Launch conditions, 8
 specifying, 648–50
 LaunchConditions table, 648
 Localized install run-time architecture, 201–7
 Basic MSI project, 206–7
 Standard project, 202–5

M

Maintenance install run-time architecture,
 189–96
 Basic MSI project, 195–96
 Standard project, 190–95
 Maintenance operations, 13, 14
 Media table, 122
 Menus
 File menu, 38
 Tools menu, 39–42
 View menu, 38–39
 Merge module project, 922–25
 Merge modules, 159–60, 159–60
 Merge Modules view, 925–28
 Microsoft Office, 83
 MIME types. *See* File associations
 MSI file, 88–90

N

NT services, 874–905
 control component - Component Wizard,
 893–99
 controlling, 881
 how they work, 874–79
 installation component - Component Wizard,
 882–93
 installing, 881
 interactive, 901–2
 SCM database, 880
 user account installation, 903–5
 NUMBER data type, 307–18

O

OBJECT data type, 323–24
 ODBC. *See* Open Database Connectivity
 ODBC Resources view, 915–19
 Open a Project view, 35–37

Open Database Connectivity, 910–21
 overview, 910–15
 Options dialog
 File Locations, 40–42
 Orca, 98, 99, 105, 112, 115, 159, 161

P

Packed GUID, 798, 799, 800, 801
 Patch packages, 161
 Patching, 16
 Path variables, 78–81
 Path Variables view, 257–64
 Project location
 setting, 40–42
 Project Wizard
 Application Features dialog, 55–59
 Application Files dialog, 59–64
 application information, 50
 Basic MSI project, 73–77
 Company Information dialog, 54
 Create Shortcuts dialog, 64–67
 creating new project, 48
 Dialogs dialog, 69–70
 feature destination, 57–59
 opening a project, 48
 project type, 49
 Registry Data dialog, 67–69
 Setup Languages dialog, 55
 Software Updates dialog, 52–53
 Wizard Summary dialog, 70
 Property Manager view, 264–66
 Property table, 96, 114, 126, 127, 146, 149,
 152, 158

Q

Qualified components, 847–48

R

RadioButton table, 132
 Registry, 12
 creating registry entries, 567–87
 REG file, 68–69
 Registry table, 569–73
 RemoveRegistry table, 573–74
 removing registry entries, 584–87
 Registry Entry tables
 definition, 106
 schema, 122–23

INDEX

- Registry table
 - schema, 569–73
- Registry view, 575–80
 - defining registry entries, 584
- RegLocator table
 - schema, 625–26
- Release Wizard, 279–91
- Releases
 - build location, 99
 - build output window, 71
- RemoveIniFile table
 - schema, 606–8
- RemoveRegistry table
 - schema, 573–74
- return* statement, 376–77

- S**
- Scanning for dependencies, 830–40
 - filtering files, 832–34
 - Visual Basic projects, 834–40
- Script libraries
 - adding functions to the Function Wizard, 428–31
 - creating, 408–28
- Searching target system
 - for files using InstallScript
 - specified location, 643–44
- Searching for
 - applications, 10–11
- Searching target system, 618–48
 - AppSearch table, 622
 - CompLocator table, 624
 - DrLocator table, 627–28
 - for files using InstallScript, 641–44
 - all fixed drives, 641–43
 - for files using tables, 630–34
 - all fixed drives, 630–31
 - specified location, 633–34
 - specified path, 631–33
 - for folders using InstallScript, 644–47
 - of existing file, 645–46
 - specified in INI file, 646–47
 - for folders using tables, 635–38
 - of existing file, 635–37
 - specified in INI file, 637–38
 - for registry value using InstallScript, 647–48
 - for registry value using tables, 638–39
 - IniLocator table, 626–27
 - overview, 619–28
 - RegLocator table, 625–26
 - Signature table, 622–24
 - Sequences view, 269–70
 - Setup Design view, 266–68
 - Setup Types view, 242–43
 - Setup.ini file, 166–71
 - SFP. *See* Windows File Protection
 - Shell object, 549
 - Shortcut table, 134–35
 - Shortcuts, 12, 251–57
 - creating in Project Wizard, 64–67
 - MSI, 66
 - standard, 66
 - Signature table
 - schema, 622–24
 - SQL, 96, 680, 686, 689, 692, 694, 696
 - Standard actions, 125
 - Standard project, 88
 - Advanced Views, 257–78
 - basic dialog template, 741–52
 - Configure the Target System, 251–57
 - custom dialogs, 738–73
 - default user interface, 724–31
 - dialog function for basic dialog template, 752–57
 - Dialogs view, 714–19
 - compiling resource files, 719–21
 - modifying user interface, 731–37
 - Organize Your Setup, 220–43
 - Prepare for Distribution, 278–93
 - silent install for custom dialogs, 770–73
 - Specify Application Data, 243–51
 - using dialog templates, 761–70
 - stdcall, 437, 442, 446
 - stdcall keyword, 434, 460, 733
 - STRING data type, 318–22
 - String ID
 - curly braces, 52
 - usage, 54
 - Structure data type, 327–30
 - Subscription, 132, 792–95
 - Summary Information Stream, 92–94
 - Application Name property, 104
 - Author property, 103
 - Category property, 101
 - Character Count property, 103
 - Codepage property, 104
 - Comments property, 103
 - Date Last Saved property, 104
 - Date of Creation property, 104
 - Developer Art application, 100–105

Keywords property, 101
 Last Printed property, 104
 Last Saved By property, 103
 Page Count property, 101–2
 Revision Number property, 103–4
 Security property, 104–5
 Source property, 103
 Subject property, 101
 Template property, 101
 Title property, 100–101
 Word Count property, 102–3
switch statement, 366–68
 System File Protection. *See* Windows File Protection

T

TCO. *See* Total Cost of Ownership
 TextStyle table, 132
 Toolbars
 customize, 39–40
 Standard, 42–46
 Tools menu, 39–42
 Top-level actions, 124–25, 138, 139, 143, 155
 Total Cost of Ownership, 84
 Transforms, 17, 160–61
 Transitive components, 846–47
try statement, 452–53

U

until statement, 374–75
 Upgrades, 16
 User interface
 Basic MSI project, 774–95
 custom dialogs in Standard project, 738–73
 default for Standard project, 724–31
 dialog basics, 704–13
 modifying default Standard project, 731–37
 serial number input in Basic MSI project, 786–92
 subscription in Basic MSI project, 792–95
 User Interface, 10–11
 User Interface tables
 ActionText table, 130–31

Binary table, 131
 Control table, 131
 ControlCondition table, 131
 ControlEvent table, 131
 definition, 106
 Dialog table, 131
 Error table, 132
 EventMapping table, 132
 RadioButton table, 132
 schema, 128–32
 TextStyle table, 132
 UIText table, 132

V

VARIANT data type, 322–23
 View menu, 38–39

W

WFP. *See* Windows File Protection
while statement, 371–73
 Windows 2000, 18
 Windows 95, 8, 9, 19, 20, 23, 28
 Windows 9x, 8
 Windows File Protection, 843–45
 Windows Installer
 automation interface, 464–83
 command line, 138–41
 database, 94–96
 design concepts, 88
 Directory table resolution, 146–50
 MSI file, 88–90
 run-time architecture, 138–54
 SDK, 98
 Summary Information Stream, 94
 user interface levels, 139–40
 Windows Installer authoring tools, 23–28
 Windows Script Host objects, 533–49

Z

ZAW. *See* Zero Administration Windows
 Zero Administration Windows, 84

InstallShield Press

End-User License Agreement

READ THIS. You should carefully read these terms and conditions before opening the software packet(s) included with this book ("Book"). This is a license agreement ("Agreement") between you and InstallShield Press ("ISPRESS"). ISPRESS is a division of InstallShield Software Corporation. By opening the accompanying software packet(s), you acknowledge that you have read and accept the following terms and conditions. If you do not agree and do not want to be bound by such terms and conditions, promptly return the Book and the unopened software packet(s) to the place you obtained them for a full refund.

1. License Grant. ISPRESS grants to you (either an individual or entity) a nonexclusive license to use one copy of the enclosed software program(s) (collectively, the "Software") solely for your own personal or business purposes on a single computer (whether a standard computer or a workstation component of a multi-user network). The Software is in use on a computer when it is loaded into temporary memory (RAM) or installed into permanent memory (hard disk, CD-ROM, or other storage device). IS PRESS reserves all rights not expressly granted herein.

2. Ownership. ISPRESS is the owner of all right, title, and interest, including copyright, in and to the compilation of the Software recorded on the disk(s) or CD-ROM ("Software Media"). Copyright to the individual programs recorded on the Software Media is owned by the author or other authorized copyright owner of each program. Ownership of the Software and all proprietary rights relating thereto remain with ISPRESS and its licensors.

3. Restrictions On Use and Transfer.

- a) You may only (i) make one copy of the Software for backup or archival purposes, or (ii) transfer the Software to a single hard disk, provided that you keep the original for backup or archival purposes. You may not (i) rent or lease the Software, (ii) copy or reproduce the Software through a LAN or other network system or through any computer subscriber system or bulletin-board system, or (iii) modify, adapt, or create derivative works based on the Software.
- b) You may not reverse engineer, decompile, or disassemble the Software. You may transfer the Software and user documentation on a permanent basis, provided that the transferee agrees to accept the terms and conditions of this Agreement and you retain no copies. If the Software is an update or has been updated, any transfer must include the most recent update and all prior versions.

4. Restrictions on Use of Individual Programs. You must follow the individual requirements and restrictions detailed for each individual program in Appendix E "What's on the CD-ROM" of this

Book. These limitations are also contained in the individual license agreements recorded on the Software Media. These limitations may include a requirement that after using the program for a specified period of time, the user must pay a registration fee or discontinue use. By opening the Software packet(s), you will be agreeing to abide by the licenses and restrictions for these individual programs that are detailed in **Appendix E** and on the Software Media. None of the material on this Software Media or listed in this Book may ever be redistributed, in original or modified form, for commercial purposes.

5. Limited Warranty.

- a) ISPRESS warrants that the Software and Software Media are free from defects in materials and workmanship under normal use for a period of sixty (60) days from the date of purchase of this Book. If ISPRESS receives notification within the warranty period of defects in materials or workmanship, ISPRESS will replace the defective Software Media.
- b) ISPRESS, LICENSORS, THE AUTHOR AND RELATED PARTIES DISCLAIM ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, WITH RESPECT TO THE SOFTWARE, THE PROGRAMS, THE SOURCE CODE CONTAINED THEREIN, AND/OR THE TECHNIQUES DESCRIBED IN THIS BOOK. ISPRESS DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE SOFTWARE WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE SOFTWARE WILL BE ERROR FREE.
- c) This limited warranty gives you specific legal rights, and you may have other rights that vary from jurisdiction to jurisdiction.

6. Remedies.

- a) ISPRESS's, licensor's, author's and related parties' entire liability and your exclusive remedy for defects in materials and workmanship shall be limited to replacement of the Software Media, which may be returned to ISPRESS with a copy of your receipt at the following address: InstallShield Press, Attn.: *Getting Started with InstallShield Developer and Windows Installer Setups*, InstallShield Software Corp, 900 N. National Parkway, Ste. 125, Schaumburg, IL 60173, or call 1-847-466-4000. Please allow three to four weeks for delivery. This Limited Warranty is void if failure of the Software Media has resulted from accident, abuse, or misapplication. Any replacement Software Media will be warranted for the remainder of the original warranty period or thirty (30) days, whichever is longer.
- b) In no event shall ISPRESS, licensor, the author, or related parties be liable for any damages whatsoever (including without limitation damages for loss of business profits, business interruption, loss of business information, or any other pecuniary loss) arising

from the use of or inability to use the Book or the Software, even if ISPRESS has been advised of the possibility of such damages.

- c) Because some jurisdictions do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitation or exclusion may not apply to you.

7. U.S. Government Restricted Rights. Use, duplication, or disclosure of the Software by the U.S. Government is subject to restrictions stated in paragraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause of DFARS 252.227-7013, and in subparagraphs (a) through (d) of the Commercial Computer- Restricted Rights clause at FAR 52.227-19, and in similar clauses in the NASA FAR supplement, when applicable.

8. General. This Agreement constitutes the entire understanding of the parties and revokes and supersedes all prior agreements, oral or written, between them and may not be modified or amended except in a writing signed by both parties hereto that specifically refers to this Agreement. This Agreement shall take precedence over any other documents that may be in conflict herewith. If any one or more provisions contained in this Agreement are held by any court or tribunal to be invalid, illegal, or otherwise unenforceable, each and every other provision shall remain in full force and effect.

