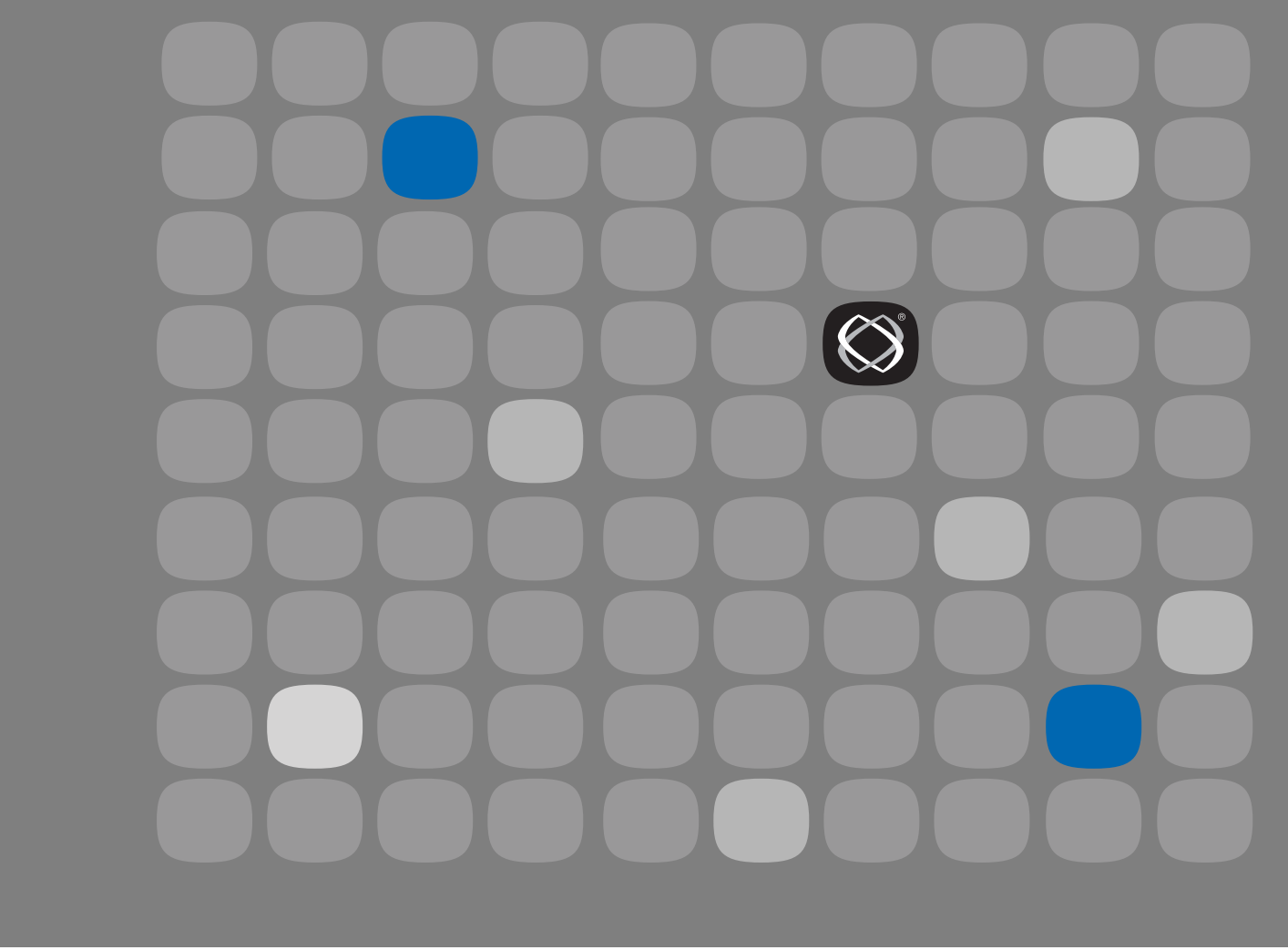


attachmate WRQ™

Reflection®

PROGRAMMING WITH REFLECTION: VISUAL BASIC USER GUIDE

WINDOWS® XP
WINDOWS 2000
WINDOWS SERVER 2003
WINDOWS 2000 SERVER
WINDOWS TERMINAL SERVER
CITRIX® METAFRAME™
CITRIX METRAFRAME XP

ENGLISH

Copyright

WRQ Reflection *Visual Basic User Guide*

Trademarks

TABLE OF CONTENTS

Chapter 1 • Overview **1**

What's in This Manual 2

Using Reflection and Visual Basic Help 2

Sample Macros 3

Chapter 2 • Getting Started **5**

Creating a Macro with the Macro Recorder 5

Running a Macro 7

Saving Macros 8

Exercises 8

Chapter 3 • Sharing and Managing Macros **15**

Distributing a Settings File that Includes Macros 15

Saving Macros in Shared (Referenced) Files 16

Exporting and Importing Visual Basic Project Files 19

Using Macros that Have Been Exported to Macro Files 20

Chapter 4 • Programming Fundamentals **23**

What is Visual Basic for Applications? 23

Basic Language Programming 25

Understanding Visual Basic Projects 26

The Reflection Object Model 27

Command Syntax 28

Named Arguments 30

Chapter 5 • Using the Visual Basic Editor **31**

The Visual Basic Editor 31

The Project Explorer 33

The Properties Window 34

The Code Window 35

Creating New Macros 37
Rules for Naming Macros 38
Editing Macros 39
Exercise 40

Chapter 6 • Creating Custom Dialog Boxes 45
Creating a New Form 45
Adding and Editing Controls 46
Writing Form Code 49
Getting User Input from Your Dialog Box 50
Opening and Closing Your Dialog Box 50
Exercises 52

Chapter 7 • Handling Errors 63
Trapping an Error 64
Resuming a Procedure After an Error 68
Inline Error Handling 69
Information About Error Codes 70

Chapter 8 • Communicating with Other Applications 71
Understanding Automation 71
Controlling Other Applications from Reflection 72
Controlling Reflection from Other Applications 74
Using CreateObject 76
Using GetObject 77
Using Reflection Predefined Constants in Other Applications 79

Chapter 9 • Using Events 81
Automation Events 81
Methods and Properties for Managing Events 83

Chapter 10 • Managing Connections to IBM Hosts 85
Commands for Connecting and Disconnecting 85
Using Connect Macros 85
Configuring Connection Settings 87

Chapter 11 • Managing Connections to HP, UNIX, OpenVMS, and Unisys Hosts 91
Commands for Connecting and Disconnecting 92
Using Connect Macros 94
Configuring Connection Settings 95
Managing Modem Connections 103
Handling Connection Errors 105

Chapter 12 • Reflection Basic Support 109
Running Reflection Basic Scripts 109
Displaying the Script Menu 110
Comparing Reflection Basic to Visual Basic 111
Why Use Visual Basic? 112
The Reflection Object Name (Application vs. Session) 113

Index 115

CHAPTER 1

Overview

This book is an introduction to macro recording and programming using the following Reflection applications:

Reflection for HP
Reflection for UNIX and OpenVMS
Reflection for ReGIS Graphics
Reflection for IBM

This chapter includes the following information:

- The contents of this manual
- How to use the Reflection and Visual Basic Editor Help menus to get complete information about programming with Reflection
- Information about sample macros

What's in This Manual

This book explains how to use Visual Basic for Applications (VBA) in Reflection and also provides information about Reflection's continuing support for Reflection Basic, a scripting language that shipped with earlier versions. It does not include reference information about Reflection's methods, properties, and events—the commands you use to control Reflection programmatically. To view reference information about these commands using Reflection's programming Help, click the Contents tab and open the book labeled Language Reference. (See page 2 for instructions on how to view this Help.)

Using Reflection and Visual Basic Help

You can use the Reflection and Visual Basic Editor Help menus to get detailed information not covered in this book. Two Help systems are available: Reflection's Programming Help and Microsoft's Visual Basic Help.

Viewing Reflection's Programming Help

Reflection's Help includes information about how to use Visual Basic for Applications in Reflection. It also includes a complete reference to Reflection-specific programming commands. To open the Reflection Programming Help:

- Open Reflection's **Help** menu, point to **Advanced Topics**, and click **Programming** (click **Programming with VBA** in Reflection for IBM).

Viewing Microsoft's Visual Basic Help

Microsoft's Visual Basic Help provides information about how to use the Visual Basic Editor and about the programming language commands that are common to all Visual Basic implementations.

To open the Visual Basic editor and view the Microsoft Help:

1. On Reflection's **Macro** menu, click **Visual Basic Editor**.
2. On the Visual Basic Editor's **Help** menu, click **Microsoft Visual Basic Help**.

Sample Macros

Settings files that contain sample macros are available to help you develop your own macros.

If you are using Reflection for IBM, you must do a Custom or Complete installation to get the sample macro files. If you install to the default folder, you'll find the sample macros in C:\Program Files\Reflection\Ibm\Samples\Vba.

If you are using Reflection for HP, Reflection for UNIX and OpenVMS, or Reflection for ReGIS Graphics, you can find sample macros on our web site. Refer to technical note 1536 (available at http://support.wrq.com/techdocs/1536.html) for more information.

CHAPTER 2

Getting Started

Reflection macros allow you to simplify and automate routine tasks you perform using Reflection. For example, you might create a macro that logs onto a host and navigates to a particular host screen. Reflection provides two ways for you to create a macro: the macro recorder and the Visual Basic Editor. This chapter describes how to use the macro recorder see Chapter 4 (page 31) for information about creating and editing macros with the Visual Basic Editor.

Topics covered in this chapter include:

- Using the Reflection macro recorder to create new macros
- Running macros
- Saving macros

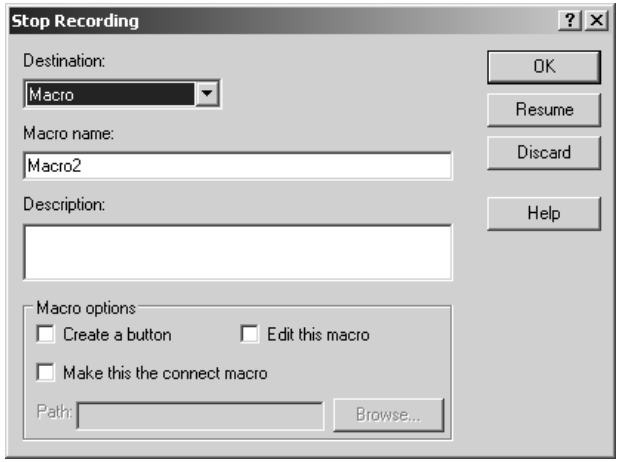
Creating a Macro with the Macro Recorder

The Reflection macro recorder lets you capture the actions you perform in Reflection. While the recorder is on, Reflection records your actions as a set of commands known as a macro. When you run a recorded macro, the sequence of commands you recorded is repeated.

To create a macro using the macro recorder:

1. On the **Macro** menu, click **Start Recording**. This starts the macro recorder.
2. Perform the actions that you want included in the macro.

3. On the **Macro** menu, click **Stop Recording**. This opens the Stop Recording dialog box.

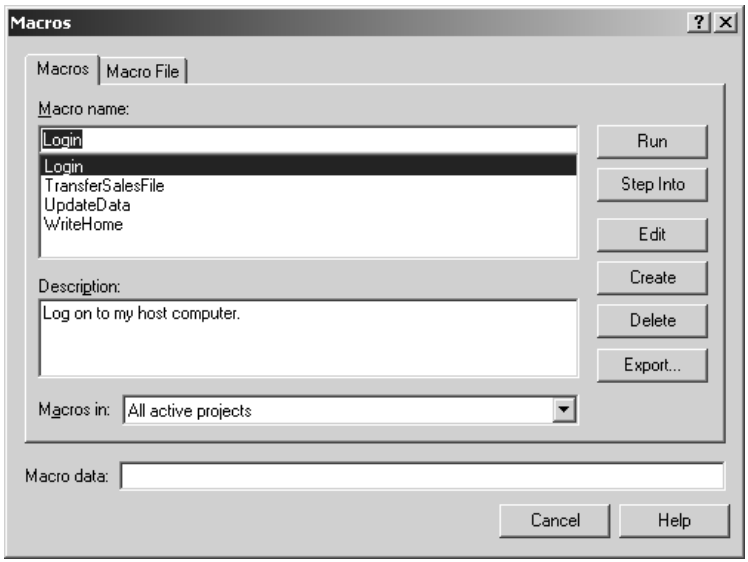


4. In the Stop Recording dialog box:
   - Leave **Destination** set to **Macro**.
   - Enter a name for your macro in the **Macro name** box. Macro names cannot include spaces and must begin with a letter. See page 38 for more information about naming macros.
   - Enter an optional description for your macro in the **Description** box.
5. Click **OK**.
6. At this point, you can run and test your macro, but it is not yet saved. If the macro works as desired, save your settings file to save the macro.

Running a Macro

When you open a settings file, all the macros created and saved in that file are available. To run a macro:

1. On the Macro menu, click Macros. This opens the Macros dialog box.



2. Type the name of the macro you want to run or select it from the list.
3. Click Run.

When a macro is running, most menu items are disabled. You can stop a running macro by clicking Stop Macro on the Macro menu.

Note: You can use the Macro File tab in the Macros dialog box to run macros that have been saved to external macro files (*.rma). (See page 20 for more information about macros in macro files.)

Saving Macros

When you first create a macro, you can run it and edit it, but it is not yet saved to your settings files. Macros are saved when you save your settings file. Either of the following actions saves both your macros and your current settings. The macros you save are available each time you open the settings file.

- In Reflection, click Save or Save As on the File menu.
- In the Visual Basic Editor, click Save <current settings file> on the File menu.

Exercises

You can use a recorded login macro to simplify host connections by automating the steps you need to take each time you make a connection. The following step-by-step exercises demonstrate how to do this.

Do Exercise 1 if you are using:

Reflection for IBM

Do Exercise 2 (page 11) if you are using:

Reflection for HP
Reflection for UNIX and OpenVMS
Reflection for ReGIS Graphics

Exercise 1: Creating a Login Macro with Reflection for IBM

This exercise demonstrates how to create a login macro using Reflection for IBM.

Steps 1-5 configure a connection to a demonstration host and create a MacroDemo settings file that you can use for saving your practice macros. If you prefer to connect to an actual host, open (or create) a settings file that is configured to connect to that host and go directly to step 6.

Creating the MacroDemo Settings File

In steps 1-5, you'll configure and test a connection, then save a settings file.

1. Open a new (untitled) Reflection session. (You can use the **New Session** command on the File menu to open a new untitled session.)
2. On the Connection menu, click Session Setup. Under **Session**, set **Type** to either **5250 Terminal** or **3270 Terminal**. Under **Transport**, set **Type** to **Demonstration**. Select the **Simulation filename** for the session type you selected. Click OK to close the Session Setup dialog box.
3. Before you record a login, you should test your connection to the host. To do this, use the **Connect** command on the **Connection** menu, or click the Connect/ Disconnect toolbar button. Type a user name and password, then press Enter to log on. For a demonstration host, you can use any login name and password.
4. After you have made a successful connection, disconnect from the demonstration host by clicking **Disconnect** on the Connection menu.
5. On the File menu, click Save As. Type *MacroDemo* in the **File name** box, then click Save.

Recording the Login Macro

In steps 6-9, you'll record the Login macro.

6. On the **Macro** menu, click **Start Recording**. The actions you perform next will be recorded and saved to a macro.

7. Connect to the host by clicking the **Connect** command on the **Connection** menu or by clicking the Connect/Disconnect toolbar button.

   Note: Reflection will also connect to your host if you press the Enter key when you are disconnected. Don't use this shortcut when you are recording connections because this keystroke is also recorded.

8. Type a user name and password, then press Enter to log on. Even if you are using a demonstration host, you should enter some text for a password; although you can make a demonstration connection if you leave this blank, your macro won't accurately simulate an actual host connection.

   Note: Unless you change the value of the Record Passwords setting (or the **RecordPasswords** property), Reflection will not include your password in any recorded macro.

9. When you have completed your login procedure, click **Stop Recording** on the **Macro** menu (or click the Start/Stop recording button on the toolbar). This opens the Stop Recording dialog box.

10. Type Login in the **Macro name** box and select the **Make this the connect macro** check box. Click OK to finish recording.

Testing the Login Macro

Steps 11 and 12 test the macro you just created.

11. Disconnect from the host.

12. Make a new connection. Because the macro you recorded is a connect macro, it will run automatically as soon as the connection is made. You should see a prompt for your password (which was not recorded). After you enter a password, the Login macro will complete execution and you should see the host prompt that indicates that you are successfully logged on.

Saving the Login Macro

Step 13 saves the macro you just created; macros are not saved until you save your settings file.

13. Open the **File** menu and save your current settings file.

Viewing the Login Macro Code

Do steps 14 and 15 if you want to see the **code that the macro recorder created:**

14. **On the Macros menu, click** Macros.

15. In the list of macros, select the Login macro you just created, then click **Edit**.

You'll see the Visual Basic Editor with the Login macro displayed in the Code window.

Exercise 2: Creating a Login Macro with Reflection for HP, UNIX and OpenVMS, and ReGIS Graphics

The following exercise demonstrates how to create a login macro using Reflection for HP with NS/VT, Reflection for UNIX and OpenVMS, or Reflection for ReGIS Graphics.

Steps 1-5 in this exercise configure a connection to a demonstration host and create a MacroDemo settings file that you can use for saving your practice macros. If you prefer to connect to an actual host, open (or create) a settings file that is configured to connect to that host and go directly to step 6.

Creating the MacroDemo Settings File

In steps 1-5, you'll configure and test a connection, then save a settings file.

1. Open a new (untitled) Reflection session. (You can use the **New Session** command on the **File** menu to open a new untitled session.)

2. On the Connection menu, click Connection Setup. Under **Connect Using**, click **Network**, then select **DEMONSTRATION** from the network list. Use the **Host Type** list to select a demonstration host type. Click **OK** to close the Connection Setup dialog box.

3. Before you record a login, you should test your connection to the host. To do this, press Enter (or use the Connect command on the Connection menu) and enter appropriate responses to the host prompts. For a demonstration host, you can use any login name and password.

4. After you have made a successful connection, log off the host. The following commands log off Reflection demonstration hosts:

   HP 3000: bye

   OpenVMS: logout

   UNIX: exit

5. On the **File** menu, click Save. Type *MacroDemo* in the **File name** box, then click **Save**.

Recording the Login Macro

In steps 6-9, you'll record the Login macro.

6. On the **Macro** menu, click **Start Recording**. This turns on the macro recorder. You'll see a small toolbar with two buttons: You can use the left button to stop recording or the right one to pause recording.

7. Connect to the host and enter responses to the host prompts (see step 3). If you are using a demonstration host, you should enter text in response to the password prompt; although you can make a demonstration connection if you leave this blank, your macro won't accurately simulate an actual host connection.

   Note: Unless you change the value of the Save Passwords setting (or the **SavePasswords** property), Reflection will not include your password in the recorded macro.

8. On the **Macro** menu, click **Stop Recording** (or click the Stop Recording button on the Recording toolbar). This opens the Stop Recording dialog box.

9. Because the steps you recorded included making a connection, Reflection automatically suggests the default name *Login* for your macro and selects the **Make this the connect macro** check box. Click OK to accept these defaults.

Testing the Login Macro

Steps 10 and 11 test the macro you just created.

10. Log off from the host. (See step 4).
11. Press Enter to make a new connection. Because the macro you recorded is a connect macro, it will run automatically as soon as the connection is made. You should see a prompt for your password (which was not recorded). After you enter a password, the Login macro will complete execution and you should see the host prompt that indicates that you are successfully logged on.

Saving the Login Macro

Step 12 saves the macro you just created; macros are not saved until you save your settings file.

12. On the **File** menu, click **Save** <settings file name>.

Viewing the Login Macro Code

Do steps 13 and 14 if you want to see the code that the macro recorder created:

13. On the **Macro** menu, click **Macros**.
14. Select Login, then click **Edit**.

You'll see the Visual Basic Editor with the Login macro displayed in the Code window.

CHAPTER 3

Sharing and Managing Macros

Reflection supports a number of strategies for sharing and managing macros. This chapter provides an overview of the following options:

- Save all macros directly to a settings file and distribute this file to end users.
- Maintain some or all of your macros in shared (referenced) settings files. You can use the default SharedMacros file and/or configure references to other settings files.
- Export Visual Basic project files and have other users import these files into their Reflection settings files.
- Use Reflection macro files (*.rma) to distribute individual macros to other users.

Distributing a Settings File that Includes Macros

You can create a single settings file that contains all of your settings and macros and distribute this file to end users.

Advantages of putting all macros and settings in a single settings file:

- Files of this kind are easy to create; Reflection saves new macros by default to the current settings file.
- Files of this kind are easy to distribute.
- When settings and macros are saved in the same file, it is easy to ensure that Reflection is correctly configured to run your macros. For example, you can configure and save the connection settings that are required for a connection macro to work correctly.

Saving Macros in Shared (Referenced) Files

Visual Basic references allow users to have access to macros that are not saved in their settings files. Reflection settings files automatically include a reference to the default SharedMacros file. You can also modify any Reflection settings file to include references to other settings files.

Advantages of using references for sharing macros

- You can maintain your macros in a centrally located file (or files). Changes and updates made to the referenced file need to be made only once. When users launch a settings file that references the centrally located file, they automatically get the latest version of your macros.
- Changes you make to referenced files have no effect on macros users have developed and saved to their own settings files.
- Referenced projects can have different protection levels from the settings file project. This means that administrators can provide functionality without making code visible to end users.

The SharedMacros File

One strategy for using referenced settings files is to save macros to the default SharedMacros file.

Advantages of putting shared macros in the default SharedMacros file

- Macros saved to the SharedMacros file are automatically included in every new Reflection session.
- You can customize Reflection to look for the SharedMacros file in any location you choose.

Working with the SharedMacros File

Use any of the following techniques to add or edit a macro in the SharedMacros file:

- From any Reflection settings file, open the Macros dialog box. On the Macros tab, set **Macros in** to **SharedMacros**.
- From any Reflection settings file, open the Visual Basic Editor, then use the Project window to browse to the SharedMacros project.
- Open the SharedMacros settings file directly and use it to create and edit macros.

Warning: To avoid losing changes you make to the SharedMacros settings file, be sure that you are editing it from only one copy of Reflection, and that no one else is working with the file at the same time.

The SharedMacros file location

Reflection sessions look for the SharedMacros file in the location specified by a setting called **Folder for the Shared Macros Settings File**. The default value (%personalfolder%\Attachmate\Reflection\) specifies the Reflection user folder.

Note: The SharedMacros project is visible in the Visual Basic Editor whenever you launch Reflection, but the SharedMacros file is not created until you add content to this project and save your changes.

You can modify **Folder for the Shared Macros Settings File** to specify any location available to other Reflection users. You can use UNC paths or URLs to identify shared network locations. Use either of the following techniques:

- In individual Reflection sessions, open the **Setup menu**, select **View Settings**, search for **Folder for the Shared Macros Settings File**, then enter a new value under Setting details. Save the settings file to keep this change.
- Specify a new default for this setting using Reflection's Group Policy support or the Reflection Profiler. (See the *Reflection System Administrator's Guide* for more information about these tools).

Adding Additional References

An additional option for using referenced settings files is to save your macros to any Reflection settings file, then add a reference to that settings file in other Reflection settings files.

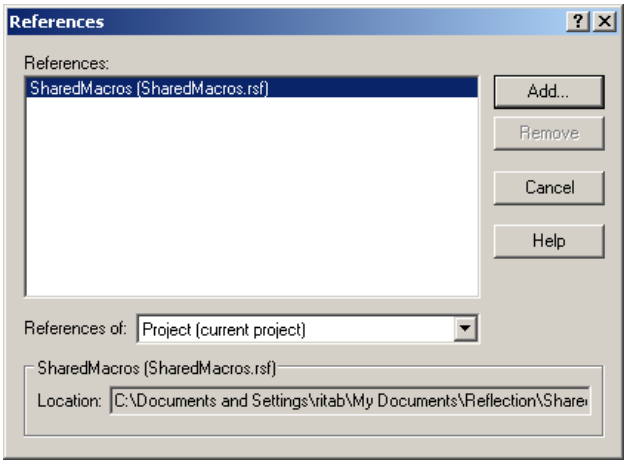
Advantages of adding additional references

- You can add references to as many files as you need. This allows you maximum flexibility in organizing your projects and distributing them to user groups.

Adding and Removing References

To add a reference:

1. On the Macro menu, select **Advanced** > **References**. This opens the References dialog box.



2. Click **Add.**
3. In the Add Reference dialog box, identify the project file you want to reference. You can browse to locate a file, or enter a UNC path or URL that points to the file location.

4. Click OK.

   If the file is a valid file, you will see a message indicating the the reference was successfully added and the referenced project will be added to References list.

5. Save your settings file to save this change.

To remove a reference:

1. On the Macro menu, select **Advanced** > **References**.
2. Under **References**, select the project you want to remove, and click **Remove**.

Note: You cannot remove the default reference to the SharedMacros project.

Exporting and Importing Visual Basic Project Files

An additional option for sharing Visual Basic code is to export code modules, classes, and/or forms; then have other users import these files into their Reflection settings files. You can use the Visual Basic Editor to export code modules (to *.bas files), classes (to *.cls files), and/or forms (to *.frm/*.frx files). Reflection users can import these files into Reflection settings files using **Macro** > **Advanced** > **Import Visual Basic File**.

Advantages exporting and importing VBA project files:

- By exporting and importing project files, you can share code between different Reflection applications, or between Reflection and other applications that use Visual Basic.
- You can export and import individual code modules or forms to add new functionality to an existing Reflection settings file without altering existing macros.

Using Macros that Have Been Exported to Macro Files

Although Reflection macros are generally part of a Visual Basic Project, you can also export some individual macros to individual Reflection macro files (*.rma) and distribute these files to end users. (See page page 21 for a comparison between macros that have been exported to separate macro files and macros that are part of Reflection settings files.) Reflection users can use the Macros dialog box (Macro File tab) to run macros that are in macro files. You can also customize Reflection features such as events, toolbar buttons, menu commands, and keyboard mapping to use macros that have been exported to macro files.

Limitations to using macro files

- You cannot use macro files to run macros that use forms or call other subroutines in your Visual Basic Project. For example, any recorded macro will run successfully after it has been exported to a macro file because all the code in any recorded macro is contained within a single subroutine. However, if you have created another macro (MacroB) that includes a call to your recorded macro (MacroA), MacroB will not run correctly after being exported to a macro file because the code in MacroA is not included in the exported file.

Advantages of using macro files

- Macro files are easy to create and distribute.
- Macro files can add new functionality to a Reflection session without altering the existing macros.

Understanding Reflection Macro files (*.rma)

The following table summarizes differences between macros that have been saved in settings files (the default) and macros that have been exported to Reflection macro files: (*.rma).

Macros in Settings Files	Macros in Macro Files (*.rma)
Settings files can contain any number of macros.	Each macro file contains exactly one macro.
Settings files fully support all Visual Basic project functionality, including user forms.	Macros in macro files are limited to a single subroutine; and cannot call other routines or user forms.
The Reflection applications that support VBA each save settings files using an application-specific file extension (*.r1w, *.r2w, *.r4w, *.rsf).	All supporting Reflection applications export macro files using the same file extension (*.rma). Exported macros may contain commands that are not supported in other Reflection applications.
A macro that is saved in a settings file is always available to any custom action (such as a toolbar button) that is configured to run the macro.	If a custom action is configured to use a macro file, you must ensure that the macro file is in the specified location.

For more information about creating and working with macro files, open the Reflection product Help and look up “Macros in macro files” in the index.

CHAPTER 4

Programming Fundamentals

This chapter introduces programming with Visual Basic for Applications and includes the following topics:

- What is Visual Basic for Applications?
- Basic Language programming
- Understanding Visual Basic projects
- The Reflection Object model
- Command Syntax

What is Visual Basic for Applications?

Visual Basic for Applications (VBA) is a combination of a programming language and a program development environment supported by many different applications (including Reflection and Microsoft Office products). You can use VBA to customize and enhance Reflection. You can also create macros that allow Reflection to interact with other applications (such as Word and Excel).

The core Visual Basic language consists of programming commands that are common to all implementations of Visual Basic. A complete language reference is available in the Microsoft Visual Basic Help; click the Contents tab and open the book labeled *Visual Basic Language Reference*. (See page 2 for instructions on how to view this Help.)

In addition to this core language, Reflection macros use Reflection-specific *methods*, *properties*, and *events* that extend the core Visual Basic language. These commands allow you to manipulate and configure Reflection sessions. A complete reference to these commands is available in the Reflection Programming Help; click the Contents tab and open the book labeled *Language Reference*. (See page 2 for instructions on how to view this Help.)

Tip: As you are editing macros, you can use context-sensitive Help to get information about Visual Basic and Reflection programming commands. Position the insertion point within the command, and press F1.

VBA uses the same language and programming environment as the stand-alone release of Visual Basic. However, stand-alone Visual Basic can be used to create executable applications that can run directly from the desktop. The Visual Basic projects you create using Reflection can only be run from a Reflection session. The entry point to a Reflection Visual Basic project is always a macro. When you create a stand-alone application, the entry point is frequently a user form.

VBA is a shared component, which means you have one copy of VBA for all applications on your system that use it. Although each application uses the same Visual Basic files, the Visual Basic Editor configuration is stored separately for each product. This means that when you open the Visual Basic Editor from Reflection, it will show your Reflection project as you last used it, even if you have used the Editor for other applications.

Basic Language Programming

Visual Basic is a modern dialect of the BASIC programming language that was first developed in the early 1960s. Visual Basic is far more powerful than the earliest versions, but many of the BASIC language commands remain unchanged. If you have no prior programming experience, you will need to become familiar with fundamental BASIC language programming concepts in order to write your own Visual Basic macros.

Note: No programming knowledge is needed if you are using the macro recorder to create your macros.

Programming language elements that are common to all implementations of BASIC include:

- Data types, variable and constants, and arrays (declared with **Dim** and **Const**)
- Operators and expressions (such as +, -, *, /, **Not**, **Like**, and **Or**)
- User-defined functions and procedures (**Sub** and **Function**)
- Control structures and loops (such as **For** ... **Next**, **Do** ... **Loop**, and **If** ... **Then** ... **Else**)

Additional concepts in VBA and stand-alone Visual Basic include:

- Objects
- Forms
- Events

The Visual Basic Help covers these and other topics under the heading *Visual Basic Conceptual Topics* in the Help Contents. (Instructions for viewing this Help are on page 2.) If you are new to BASIC programming, you may want to use a guide that is organized for beginners before you tackle the Help. Many books are available that cover the fundamentals of programming with Visual Basic and Visual Basic for Applications.

Understanding Visual Basic Projects

A macro is the entry point to a Visual Basic project. When you run a macro, Visual Basic executes the commands in that macro. Commands within a macro can use other components of the same project. This means that a macro can run other macros, display forms, or execute user-defined procedures—if these components are within the same project. In addition, macros in one project can also run macros in other referenced projects.

When you open the Visual Basic Editor, you can see the elements of your current project by using the Visual Basic Project Explorer.

The term *scope* refers to the availability of variables, constants, procedures, and forms defined in one part of a project for use by other procedures in your project. See *Scoping levels* in the Visual Basic Help for more information. (Instructions for viewing this Help are on page 2.)

The Components of a Reflection Project

A Reflection Visual Basic project contains a number of components, including Objects, Modules, Forms, and Class Modules.

Reflection Objects

The Reflection objects component contains the module that defines the methods and properties that make up the Reflection object. If you select ThisSession in the Project Explorer, the Properties window lists all the properties for Reflection's **Session** object. This list is similar to the list you see when you select **Macro syntax** in Reflection's View Settings dialog box. Changes you make in either location affect the current Reflection settings.

Modules

A Modules component is present in a project if you have created any Reflection macros or modules using the Module command in Visual Basic's Insert menu. The Modules folder in the Project Explorer lists code modules that include the programming procedures and declarations you have added to your project. When you record macros in Reflection or create new macros using the Create button in the Macros dialog box, they are placed in a module called NewMacros. You can add your own procedures to this module or create new modules. You can double-click a code module in the Project Explorer to display that module in the Code window.

Forms

A Forms component is present in a project if you have created user forms. Forms are custom dialog boxes that display information and get input from users. See Chapter 5 (page 45) for more information.

Class Modules

A Class Modules component is present in a project if you have created class modules. Class modules are a powerful programming feature that allow you to create user-defined object classes, including methods and properties for controlling these objects.

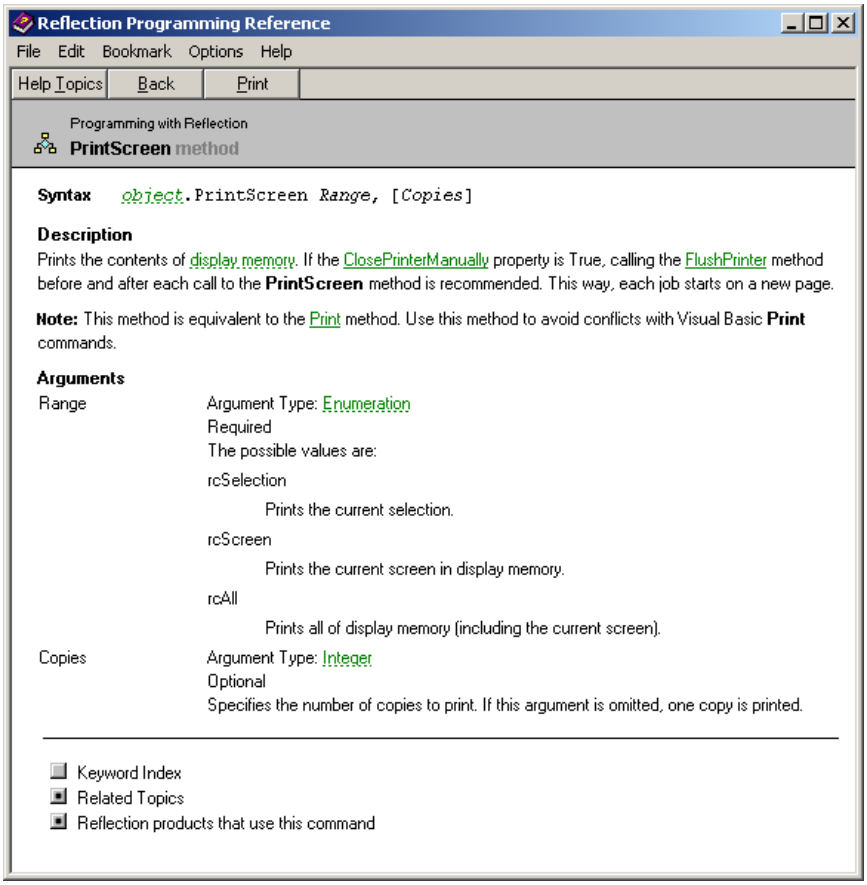
The Reflection Object Model

Reflection uses one object: **Session**. All of Reflection's methods and properties act on the **Session** object.

Note: Prior to version 7.0, Reflection used the name **Application** for the Reflection object. If you are creating or maintaining Reflection Basic scripts, continue to use **Application** for the Reflection object name.

Command Syntax

To view the Help topic for a Reflection method, property, or event, you can search for the command name in the Reflection Programming Help. (Instructions for viewing this Help file are on page 2.) You can also get command help by pressing F1 when the insertion point is positioned on a command in the Visual Basic Editor. The sample Help topic shown here is from the Reflection for UNIX and OpenVMS Help.



Every keyword Help topic includes a syntax line. Following are some guidelines for understanding and using correct command syntax:

- Items that are not shown in italics are part of the command language. You should use these in your macros exactly as they appear in the sample syntax. Items that are shown in italics are place holders. Replace these with your own values or expressions.
- Arguments must be used in the order shown in the syntax statement. The exception is if you use named arguments. (See page 30.)
- Keywords and arguments are not case sensitive.
- If an argument is enclosed in square brackets [like this], it is optional.
- Use quotation marks for string arguments when you are using literal strings, (but not when you are using string variables). Use double quotation marks for strings within other strings. For example:

  Session.SomeMethod "Say ""Hello"" to Joe"

- When you use a Reflection method or property in a Visual Basic macro, precede it with the Reflection object (**Session**) and a period, or use a **With** statement to identify **Session** as the object. You can also use just the command name (with nothing preceding it), however this can cause conflicts if you have defined variables or procedures that use the same name as a Reflection method or property. These statements are equivalent:

  Preceding the command with the Reflection object:

  ```
  Session.Connect
  ```

  Placing the command or commands between **With** and **End With** statements:

  ```
  With Session
     .Connect
  End With
  ```

  Using just the method name:

  Note: If you have declared a variable named connect, this statement will generate an error. If you have defined a Connect procedure, this statement will run that procedure.

  ```
  Connect
  ```

Named Arguments

If you use the syntax shown in the Help topic for any Reflection method, you must put your arguments in the same order as they appear in the syntax line. Use named arguments if you want to reorder the arguments, omit optional arguments, or as a means of helping identify arguments in your commands.

A named argument consists of a token to identify an argument, followed by a colon and an equal sign, and then the value for the argument:

```
Token:= ArgumentValue
```

The token name is the same as the argument name used in the syntax statement. For example, the **Transmit** method (supported in Reflection for HP, Reflection for UNIX and OpenVMS, and Reflection for ReGIS Graphics) takes two arguments:

```
Transmit String, [Options]
```

Using standard syntax, the String argument must always be given first. For example:

```
Session.Transmit "mypass", rcDecodePassword
```

Using tokens derived from the syntax line, you can modify this command to use named arguments:

```
Session.Transmit String:= "mypass", Options:= rcDecodePassword
```

Named arguments allow you to reorder arguments, so the following command is equivalent to the one above:

```
Session.Transmit Options:= rcDecodePassword, String:= "mypass"
```

For user-defined procedures, the token name is the variable name you use when you define the procedure.

CHAPTER 5

Using the Visual Basic Editor

The Visual Basic Editor allows you to modify recorded macros or to create new ones. You can use Visual Basic to create more flexible, powerful macros that include features (such as dialog boxes and conditional statements) that cannot be created using Reflection's macro recorder. The Visual Basic Editor you use in Reflection is identical to that used by many other applications (including Microsoft Office applications). This means that expertise you acquire with one product will help you develop macros in other products.

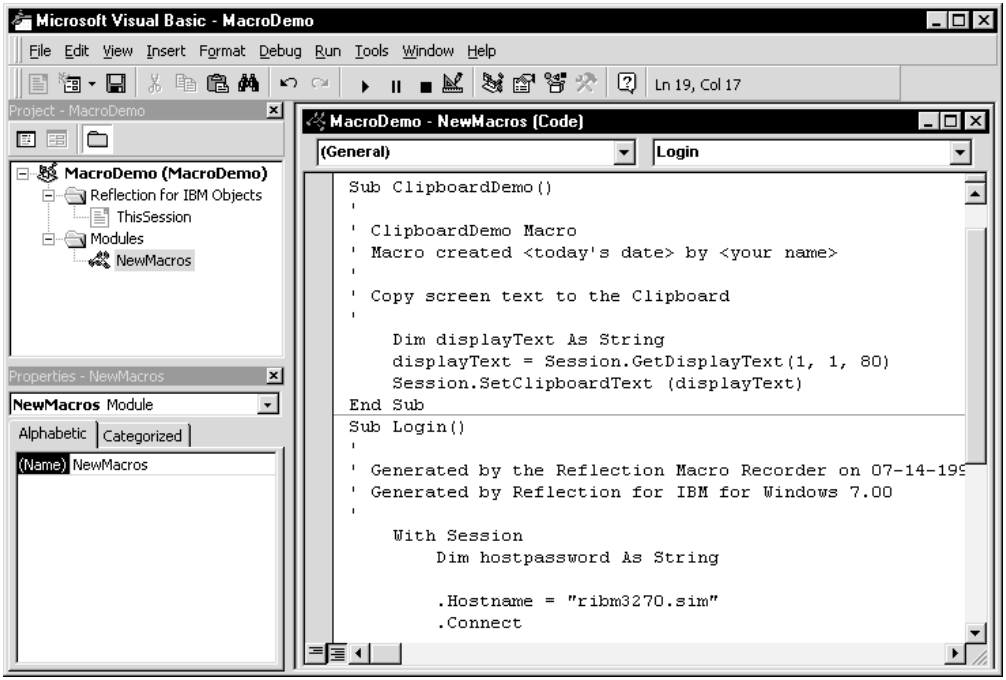
This chapter includes:

- An overview of the Visual Basic Editor
- Descriptions of the Project Explorer, the Properties window, and the Code window
- Procedures for creating and editing macros
- Rules for naming macros
- A step-by-step exercise that demonstrates how to create, test, run, and save a Reflection macro using the Visual Basic Editor

The Visual Basic Editor

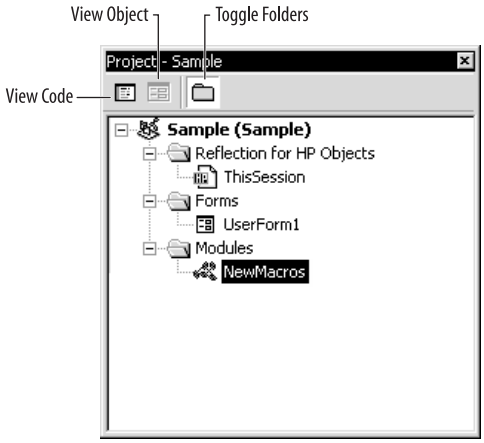
The Visual Basic Editor is an integrated development environment for writing, editing, and debugging Reflection macros. The first time you edit a Reflection macro, you'll see the Project Explorer, Properties window, and Code window. If you create user forms, you'll also work with the UserForm window. (See Chapter 5 on page 45 for more information about creating forms.) Three additional windows—Immediate, Locals, and Watch—are useful for testing and debugging.

Detailed information about each of the Editor's features is available in the Visual Basic Help; click the Contents tab and see *Visual Basic User Interface Help* and *Visual Basic How-To Topics*. (Instructions for viewing this Help are on page 2.)



The Project Explorer

The Project Explorer displays the elements of your current project. (See page 26 for information about the components of a Reflection project.)



By default, the Project Explorer arranges items in the project in related groups using a tree diagram. Click the plus sign to expand a branch of the tree, or click the minus sign to collapse that branch.

There are three buttons at the top of the Project Explorer:

- **View Code** displays the code for the currently selected object. For example, if you select the NewMacros module and click View Code, the code window opens with the insertion point in the macro you most recently edited.

  Tip: Double-clicking a module also displays the code for that module.

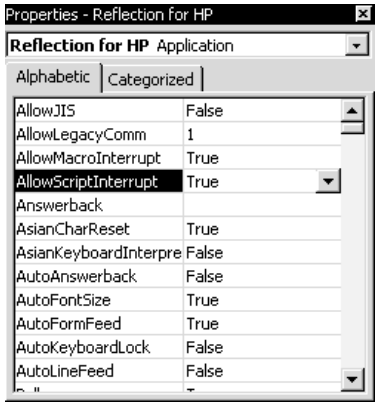
- **View Object** displays the currently selected object. For example, if you select a user form and click View Object, the UserForm window opens with that form visible. You can also display objects by double-clicking them.

- **Toggle Folders** changes the display in the Project Explorer so items are no longer arranged in related folders. When you toggle folder display off, components are listed alphabetically rather than in related groups. Click this button again to return to the default display.

The Properties Window

The Properties window lists all the properties of the currently selected object. If a code module is selected, the only thing visible in the Properties window is the module name. If a form is selected, you can use the Properties window to modify the form and its controls.

If the Reflection **ThisSession** object is selected, the Properties windows shows the current value of all of Reflection's properties.



The Code Window

The Code window displays all of the code (procedures and declarations) for a given module. Some key features of the Code window are summarized here. For more detailed information, search for *Code window* in the Visual Basic Help index. (See page 2 for instructions on how to view this Help.)

Getting Help

Context-sensitive Help is available for Visual Basic and Reflection commands. To view a Help topic, position the insertion point within a command and press F1.

Editing Code

Visual Basic provides a number of features to help you type and edit code. You can configure these using the Options dialog box. The Auto Quick Info feature displays information about command arguments as you type. (To see this, type **SetClipboardText** in the Code window, then press the spacebar.) The Auto Syntax Check feature determines whether Visual Basic checks your syntax after each line of code you type. Drag-and-drop text editing allows you to drag and drop elements within the current code and from the Code window into the Immediate or Watch windows. For more information about these and other features, click Options on the Visual Basic Editor's Tools menu, click the Editor tab, then click the Help button.

Viewing Macros

The Procedure list in the upper-right corner of the Code window allows you to quickly locate any macro or other procedure in a module. When you display this list, you see a list of items in the current module.

The buttons in the lower-left corner of the Code window determine how procedures are displayed. By default, procedures are displayed in Full Module view. In this view, all procedures are visible in a single, scrolling window, with a line between each procedure. In Procedure view, each procedure is displayed separately.

The split bar allows you to view and edit different procedures in the same module. To split the Code window, drag the split box at the top of the scroll bar (immediately above the up arrow). Drag the split bar to the top or the bottom of the window or double-click the bar to close the pane.

Creating New Macros

You can create new macros using both Reflection and the Visual Basic Editor. In Reflection, use the macro recorder if you want to create macros without writing any Visual Basic code. You can also use recorded macros as a starting point, and then edit the code in the Visual Basic Editor.

Each of the step-by-step procedures that follows creates a new macro. Follow the steps to create a new **Sub** procedure, then place your macro code between the **Sub** and **End Sub** statements of this procedure.

Creating a New Macro Using Reflection's Macros Dialog Box

To create a new macro using Reflection's Macros dialog box:

1. In Reflection, on the **Macro** menu, click **Macros**.
2. Type a macro name in the **Macro name** box.
3. Click **Create**.

Reflection automatically places new macros in the NewMacros module. You can create new macros in other modules using one of the following procedures:

Creating A New Macro Using the Visual Basic Editor's Add Procedure Dialog Box

To create a new macro using the Visual Basic Editor's Add Procedure dialog box:

1. In the Visual Basic Editor Project Explorer, double-click the module in which you want to put your new macro. This displays the Code window for that module. (By default, Reflection places macros in a module called NewMacros. To create your own modules, on the **Insert** menu, click **Module**.)
2. On the **Insert** menu, click **Procedure**.
3. Type a macro name in the **Name** box.
4. Macros are **Public Sub** procedures. These options are selected by default in the **Type** and **Scope** group boxes.
5. Click **OK**.

Creating a New Macro by Writing Code

To create a new macro by writing code:

1. In the Visual Basic Editor's Project Explorer, double-click the module in which you want to put your new macro. This displays the Code window for that module. (By default, Reflection places macros in a module called NewMacros. To create your own modules, on the Insert menu, click Module.)
2. Type **Sub** followed by a space, then type a macro name.
3. Press Enter. The Editor automatically creates an **End Sub** statement. Type code for your macro between these statements.

Rules for Naming Macros

Use the following rules when you name macros:

- Use a letter as the first character.
- You can use alphanumeric characters and the underscore character (_), but spaces and other symbols are not allowed.
- The macro name can't exceed 255 characters in length.
- Don't use any names that are the same as the Visual Basic or Reflection commands. Or, if you do use a macro name that is the same as a command, fully qualify the command when you want to use it. To do this, you need to precede the command name with the name of the associated type library. For example, if you have a macro called Beep, you can only invoke the Visual Basic **Beep** statement by using **VBA.Beep**.
- You can't repeat names within the same level of scope. This means you can't have two macros named StartUp in the same code module, but you can have two StartUp macros if they are in different code modules. To call a macro with a duplicate name that is in another code module, you must fully qualify the macro name. For example, Module1.StartUp will invoke the StartUp macro in Module1.

The naming rules described here for macros also apply to procedures, constants, variables, and arguments in Visual Basic modules.

Note: Visual Basic isn't case sensitive, but it preserves the capitalization you use when you name macros. This allows you to create macro names that are easier to read.

Editing Macros

To edit existing macros, you can find them using Reflection's Macros dialog box or using the Visual Basic Code Window.

To display a macro to edit using Reflection's Macro dialog box:

1. In Reflection, on the **Macro** menu, click **Macros**.
2. Select the macro you want to edit from the list of macros, or type the macro name in the **Macro name** box. (The macros lists shows all macros in the current settings file and any referenced files.)
3. Click **Edit**.

To display a macro to edit using the Visual Basic Editor:

1. In the Visual Basic Editor Project Explorer, double-click the module containing the macro. This displays the Code window for that module. (Any macros you create using Reflection's macro recorder or Reflection's Macros dialog box are located in a module called NewMacros.)
2. Select the macro you want to edit using the Procedures box in the upper-right corner of the Code window.

Exercise

This exercise demonstrates how to create a macro using the Visual Basic Editor. The macro you create uses Reflection methods to get text from the screen display and copy it to the Clipboard. As you type the code, you'll have a chance to see some of the Visual Basic Editor features that help simplify this process. The completed code is on page 43.

Creating the ClipboardDemo Macro

In steps 1-7, you open the Visual Basic Editor and use its editing features to create the code for the macro:

1. Open the Reflection settings file you are using for your practice macros. The macro you create in this exercise copies text from the screen display, so you should connect to a host.

2. On the Macro menu, click Macros. In the **Macro name** box, type `ClipboardDemo`. In the **Description** box, type `Copy screen text to the Clipboard.`

3. Click Create; this closes the Macros dialog box and opens the Visual Basic Editor with the following code already entered. The lines preceded by apostrophes are comments. They add useful information for someone reading the macro, but have no other effect.

```
Sub ClipboardDemo()

   ' ClipboardDemo Macro
   ' Macro created <today's date> by <your name>

   ' Copy screen text to the Clipboard

End Sub
```

4. This macro uses a string variable to hold the screen text. You can take advantage of the Visual Basic Editor's editing features to write the statement that declares this variable. Make sure the insertion point is located beneath the comment code, press Tab to indent your code, then type:

```
dim displayText as
```

   Press the spacebar, and the Editor displays a list of valid variable types. Type `s` to jump to the items on the list that start with s, then double-click the String item. The Editor automatically inserts the item you select into your code.

5. Press Enter. The insertion point will move to a new line indented at the same level as the previous line. The Editor will also format your code automatically, so that Visual Basic commands are identified by color and begin with uppercase letters. Your statement should look like this:

   ```
   Dim displayText As String
   ```

   Variable names are not case sensitive. The Editor retains the capitalization you use when you type variables. Identifying variables by using a lowercase initial letter helps distinguish them from the commands.

   Tip: For more information about the **Dim** statement, position the insertion point on **Dim** and press F1. This opens the Microsoft Help topic for this command.

6. The next line in your macro gets 80 characters of text from the host display and assigns this string value to the displayText variable. The method you use to do this depends on the Reflection product you are using. Type one of the following. (You may need to adjust the coordinates to capture text from your display):

   If you are using Reflection for HP, Reflection for UNIX and OpenVMS, or Reflection for ReGis Graphics, type:

   ```
   displayText = Session.GetText(1, 0, 1, 80)
   ```

   If you are using Reflection for IBM, type:

   ```
   displayText = Session.GetDisplayText(1, 1, 80)
   ```

   Notice that as you type code for a Reflection method, the Editor displays syntax information about this method. Press F1 when the insertion point is positioned on a method to open the Reflection Help file with more detailed information.

7. The next line in your macro uses the **SetClipboardText** method to place the display text in the Clipboard. Type the following:

   ```
   Session.SetClipboardText(displayText)
   ```

8. The last line in your macro uses a **MsgBox** statement to let you know the macro has done something. Type the following:

   ```
   MsgBox "Screen text has been copied to the Clipboard."
   ```

Your macro is done. The completed code is on page 43.

Testing the Macro

Steps 9-11 test the macro:

9. On the Visual Basic Editor's View menu, click Immediate Window. The Immediate window is a debugging tool that allows you to test code. In this exercise, you'll just use it as a scratch pad to paste the Clipboard contents.

10. Place the insertion point anywhere in the ClipboardDemo procedure you just created. You can run this procedure from the Editor using any of the following techniques:

    - Click the Run button on the Visual Basic toolbar.
    - Press F5.
    - On the Run menu, click Run Sub/UserForm.

11. Click in the Immediate window and press c+V to paste the Clipboard contents. You should see the text that was copied from your host screen.

Saving the ClipboardDemo Macro

Step 12 saves the macro you just created; macros are not saved until you save your settings file. You can save settings files using either the Visual Basic Editor or Reflection.

12. On the Visual Basic Editor's File menu, click the Save command, which identifies your current Reflection settings file (or uses Untitled if you have not created a settings file). Using this command is equivalent to using the Save command on Reflection's File menu.

The Completed ClipboardDemo Macro

Refer to the sample code for the Reflection product you are using.

Reflection for IBM

If you are using Reflection for IBM, your ClipboardDemo code should look like this:

```
Sub ClipboardDemo()
'
' ClipboardDemo Macro
' Macro created <today's date> by <your name>
'
' Copy screen text to the Clipboard
'
   Dim displayText As String
   displayText = Session.GetDisplayText(1, 1, 80)
   Session.SetClipboardText (displayText)
   MsgBox "Screen text has been copied to the Clipboard."
End Sub
```

Reflection for HP
Reflection for UNIX and OpenVMS
Reflection for ReGIS Graphics

If you are using Reflection for HP, Reflection for UNIX and OpenVMS, or Reflection for ReGIS Graphics, your ClipboardDemo code should look like this:

```
Sub ClipboardDemo()
'
' ClipboardDemo Macro
' Macro created <today's date> by <your name>
'
' Copy screen text to the clipboard
'
   Dim displayText As String
   displayText = Session.GetText(1, 0, 1, 80)
   Session.SetClipboardText (displayText)
   MsgBox "Screen text has been copied to the Clipboard."
End Sub
```

CHAPTER 6

Creating Custom Dialog Boxes

Macros use dialog boxes to display information and get user feedback. To create dialog boxes, you add forms (also called UserForms) to your project. This chapter describes the steps needed to create and edit user forms, including:

- Creating a new form
- Adding and editing controls
- Writing form code
- Getting user input from your dialog box
- Opening and closing your dialog box
- Step-by-step exercises for creating and getting user input from dialog boxes

Creating a New Form

To create a custom dialog box, insert a new form in your project following these steps:

1. Open the Visual Basic Editor.
2. On the **Insert** menu, click **UserForm**. This opens a new, blank user form. When you first create a form, the entire form is selected and you can readily modify its properties.
3. If you want to resize the form, drag the small, square resizing handles.

4. Use the Properties window to specify the properties of this form. (If it's not already visible, on the **View** menu, click **Properties Window**.) Properties you may want to change include:

   - **Name:** Forms are identified by this name in the Project Explorer. You'll use this name when you want to call the form from a macro. Visual Basic uses a default name, such as UserForm1. You can change this to a more meaningful name.
   - **Caption:** The caption appears in the title bar when the dialog box opens. Change this to a word or phrase that will help identify the dialog box to your user.

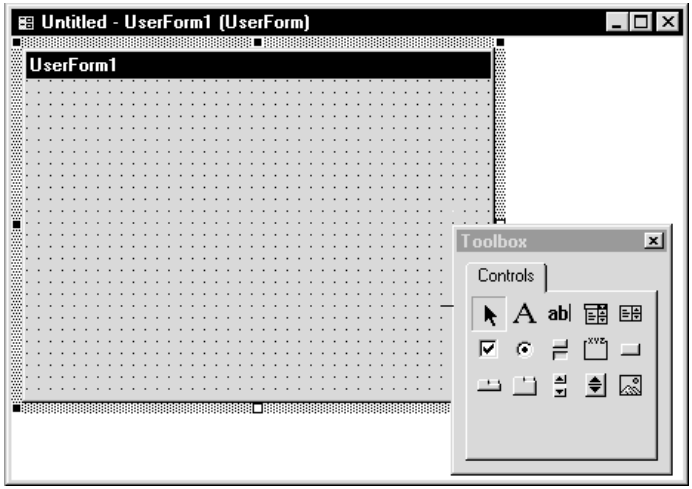
For a detailed explanation of these and other UserForm properties, look up *UserForm object* in the Visual Basic Help and click the Properties link at the top of the UserForm Object topic. (See page 2 for instructions on how to view this Help.)

Adding and Editing Controls

Controls on dialog boxes allow you to provide information and get user input. Different controls are appropriate for different purposes. Examples of some commonly used controls include:

- **CommandButton:** Allows a user to carry out an action.
- **TextBox:** Allows a user to enter text.
- **Label:** Identifies portions of the dialog box and displays information to the user.
- **ListBox:** Displays a list of items from which the user can select.
- **ComboBox:** Displays a list of items.
- **OptionButton:** Allows users to select from mutually exclusive options.
- **CheckBox:** Allows users to select yes/no options that are not mutually exclusive.
- **Frame:** Draws an outline that groups related controls.

To add or edit controls on a form, display the form and the Toolbox in the Visual Basic Editor. You can display both by double-clicking a form name in the Project Explorer. All of the available controls are shown on the Toolbox.



For a detailed explanation of each control and the properties it supports, refer to *Microsoft Forms Object Model Reference* in the Contents tab of the Visual Basic Help. (See page 2 for instructions on how to view this Help.)

For more information about using the Toolbox and designing forms, look up *Toolbox* in the Visual Basic Help Index, or see *Microsoft Forms Design Reference* in the Visual Basic Help Contents.

Adding a New Control to a Form

Use this procedure to add a new control to a form:

1. Click the command button in the Toolbox that identifies the control you want to add.

   Tip: To display a ToolTip that identifies each control, position your mouse pointer over the control and wait a few seconds.

2. Position the mouse pointer on your form where you want the new control. Click and hold down the mouse button, then drag the control until it is the size you want, or click without dragging to insert a control that uses default dimensions.

Editing Control Properties

To edit the properties of a control:

1. Click the control you want to edit; this selects that control.
2. To reposition the control, drag it to a new location. To resize it, use the small, square sizing handles that appear around the edge of the control when it is selected.
3. To change other properties, use the Properties window. (If it's not already visible, on the View menu, click Properties Window.) Properties you may want to change include:

   - **Name:** Visual Basic assigns default names like CommandButton1 and TextBox1 to new controls. These names are used to identify the control in your code. You can leave a control's name unchanged or edit it to a more meaningful name.

     Warning: If you plan on changing a control name, do it before you create an event procedure for the control. If you change a control name after you have written an event procedure, you need to edit the event procedure name as well.

   - **Caption:** Captions apply to controls, such as command buttons and labels, that are typically identified with text. Use the Caption property to specify the text that appears on the control when the dialog box opens.

Tip: You can also edit controls directly on a form. Click to select the control, then wait a few seconds and click again. Edit the text, then click outside the control to redisplay the Toolbox.

Writing Form Code

Dialog box controls frequently need to trigger appropriate actions based on a user's actions. In Visual Basic, you define event procedures to respond to user actions. An event procedure is code that is executed when a particular event occurs. For example, if you have a button labeled OK on your form, you need to write an appropriate event procedure that executes when this button is clicked.

To write event procedures for a form:

1. Display the form in the Visual Basic Editor. You can display a form by double-clicking the form name in the Project Explorer.
2. On the **View** menu, click **Code** to display the form Code window.
3. Using the Object list in the upper-left area of the Code window, select the control for which you want to write a procedure. (Controls are identified here using their **Name** property.)
4. Using the Procedure list in the upper-right area of the Code window, select the event that you want to define. For example, select Click if you want the procedure to occur when the user clicks this control. At this point, the editor will do one of the following:
   - If a procedure exists for this event, it is displayed in the Code window.
   - If no procedure exists, the Editor creates a new event procedure and places the insertion point in the procedure where you will write your code.

   Event procedures are identified with a name that includes both the control name and the event; for example, the following procedure will govern what happens when the CommandButton1command button is clicked:

   ```
   Private Sub CommandButton1_Click()
   ```

Tip: You can also double-click a control in the Form window to create an event procedure for that control. The Visual Basic Editor automatically creates a procedure using the default event for that control.

5. Write the code that you want executed when this event occurs; for example, this procedure displays a message saying "Hello World!" when a Hello command button is clicked:

   ```
   Private Sub Hello_Click()
      MsgBox "Hello World!"
   End Sub
   ```

Getting User Input from Your Dialog Box

Some actions a user takes in a dialog box change the properties of the control being used. For example, typing text in a TextBox control changes its **Text** property and clicking an OptionButton changes its **Value** property. When you want to determine what a user has done with a particular control, you can return the value of the relevant property.

For example, the following command displays the current text in the TextBox1 control:

```
MsgBox TextBox1.Text
```

This command tests to see the current state of the CheckBox1 control:

```
If CheckBox1.Value = True Then
```

If a form is loaded into memory, you can use the form properties to return information to any procedure in the same project. To do this, fully qualify the control name by including the UserForm name. For example, the following expression can be used in a macro to return the text in the *TextBox1* control located in the *UserForm1* form:

```
UserBox1.TextBox1.Text
```

Opening and Closing Your Dialog Box

Once you have created a user form, you need to add code that opens and closes this form.

Opening a Dialog Box

To open a dialog box, use the **Show** method. For example, you could use the following command in a macro to display the UserForm1 dialog box:

```
UserForm1.Show
```

If this dialog box has not yet been loaded in memory, this command loads it and displays it. You can load a dialog box into memory without displaying it using the **Load** statement. For example:

```
Load UserForm1
```

When a form is loaded, you can access information about it using form and control properties.

Tip: It is easy to display and test a dialog box while you work with it in the Visual Basic editor—just open the Debug menu and click Run Sub/UserForm.

Closing a Dialog Box

You can close a dialog box by hiding it using the **Hide** property or unloading it using the **Unload** statement. When you hide a form, it is no longer visible to the user, but you can still access information about the form by referencing the UserForm object and its control objects. When you unload a form, it is removed from memory and you can no longer access information about it.

Within a form's code, you can identify the form by using **Me** for the object name. (You can also use the **Name** property for the form.) The following example shows a procedure that is activated when a button named CloseButton is clicked. The procedure closes the dialog box without unloading it from memory.

```
Private Sub CloseButton_Click()
   Me.Hide
End Sub
```

This next procedure closes the dialog box and removes it from memory.

```
Private Sub CloseButton_click()
   Unload Me
End Sub
```

If a user closes a dialog box by clicking the Close button in the upper-right corner, the form is unloaded.

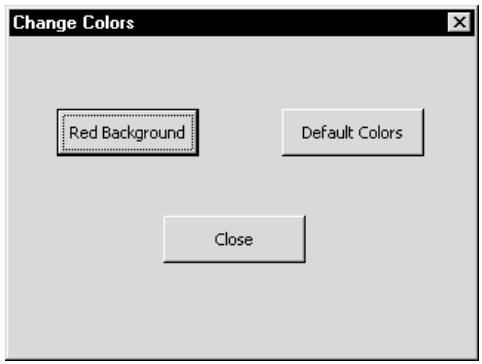
Exercises

The exercises that follow demonstrate how to use forms to create custom dialog boxes. In Exercise 1, you create a dialog box with buttons that change the color of your Reflection window. In Exercise 2, you create a dialog box that allows users to select a file on their computer and then displays the file that was selected.

Tip: These exercises are available in the Reflection Programming Help. You can use this Help file if you want to copy and paste code, rather than type it manually. See page 2 for instructions on how view this Help. Look up *Tutorial* in the index.

Exercise 1: Creating a Dialog Box

This exercise demonstrates how to create a form using the Visual Basic Editor. The macro you create opens a dialog box that changes the background color in the Reflection window. The finished Change Colors dialog box is shown in the figure.



Creating a New UserForm

In steps 1-4, you open the Visual Basic Editor and create a new UserForm.

1. Open the Reflection settings file you are using for your practice macros.
2. On the Macro menu, click Macros. In the **Macro name** box, type `ColorDemo`, then click Create.
3. If the Properties window is not already visible in the Visual Basic Editor, on the View menu, click Properties Window. You'll use this window to modify the properties of your UserForm and its controls.
4. On the Insert menu, click UserForm; this creates a new blank form. If the Toolbox window isn't open, click Toolbox on the View menu.

Designing the Dialog Box

In steps 5-12, you add custom features to the dialog box.

5. In the Properties window, double-click (**Name**) at the top of the **Properties** list. This selects the default name (UserForm1 or a similar name). Type `ColorDemoDialog` to replace the default name. This name identifies the dialog box in your project.
6. Select the **Caption** property and change it to `Change Colors`. This changes the text that shows up in the title bar of your dialog box.
7. While the Properties window is active, the Toolbox is not visible. Activate the UserForm window to display the Toolbox. (You can activate this window by clicking it, by using the Editor's Window menu, or by using the Project Explorer.)
8. Locate the CommandButton control on the Toolbox. (By pausing on a control, you can use ToolTips to identify it.) Drag this control to your dialog box, then release the mouse button. This places a default command button on your form.

9. Practice dragging this control to new locations in the form, then drag it to the position of the Red Background button in the finished dialog box.

10. Use the Properties window to change the following properties: Set **Name** to *RedButton* and set **Caption** to *Red Background*.

11. Return to the UserForm window, add a second button, and position it where the Default Colors button is in the finished dialog box. For this button, set **Name** to *DefaultButton*, and set **Caption** to *Default Colors*.

12. Add a third button where the Close button is in the finished dialog box. For this button, set **Name** to *CloseButton* and set **Caption** to *Close*.

Adding Code to the Form

In steps 13-16, you add code to the user form that controls what happens when a user clicks the command buttons. The complete code for this dialog box is on pages 56 and 57.

13. Double-click the Connect button in your user form. This opens the UserForm Code window, with the following event procedure automatically in place. The code you place between these lines executes when the user clicks the Red Background button.

```
Private Sub RedButton_Click()

End Sub
```

14. Edit this procedure to change the color of the Reflection window, using the code shown here for the Reflection product you are using:

    If you are using Reflection for HP, Reflection for UNIX and OpenVMS, or Reflection for ReGIS Graphics, use the **SetColorMap** method to change the foreground color to white and the background color to red:

```
Private Sub RedButton_Click()
   Session.SetColorMap rcPlainAttribute, rcWhite, rcRed
End Sub
```

    If you are using Reflection for IBM, use the **BackgndColor** property to change the background color to red:

```
Private Sub RedButton_Click()
   Session.BackgndColor = rcRed
End Sub
```

15. Return to the UserForm window, and double-click the Default Color button to create an event procedure for this button. The statement you add uses the **RestoreDefaults** method to restore the default colors.

```
Private Sub DefaultButton_Click()
   Session.RestoreDefaults rcColors
End Sub
```

16. Finally, create a procedure for the Close button with the following code. The **Unload** statement closes the dialog box and removes it from memory. You can use **Me** to refer to a UserForm object within your form code. *Unload Me* in this example is equivalent to *Unload ColorDemoDialog.*

```
Private Sub CloseButton_Click()
   Unload Me
End Sub
```

Testing the Dialog Box

The next step tests the dialog box before you add it to your macro.

17. To test the dialog box, activate the UserForm window, then click the Run button on the Visual Basic toolbar. Try the Red Background and Default Colors buttons, then click the Close button.

Displaying the Dialog Box from Your Macro

In steps 18 and 19, you add code that opens your dialog box to the ColorDemo Macro.

18. Double-click NewMacros in the Project Explorer to open this Code window. Position the insertion point beneath the comment code of the ColorDemo procedure. The code you enter here runs when you run the ColorDemo macro.

19. Type the following line immediately beneath the comments at the top of the procedure. This line displays the UserForm you just created.

```
ColorDemoDialog.Show
```

Testing and Saving the ColorDemo Macro

In the final steps of this exercise, you test and save your macro.

20. Return to Reflection. (You can use the Windows taskbar or click the Reflection item at the bottom of the Editor's View menu.)
21. On the Macro menu, click Macros. Select the ColorDemo macro from the list and click Run. This should display the dialog box you just created. Test and close the dialog box.
22. Save your settings file to save this macro.

The ColorDemo Dialog Box UserForm Code

Refer to the sample code for the Reflection product you are using.

ColorDemo Code for Reflection for IBM

If you are using Reflection for IBM, your UserForm code for the ColorDemo dialog box will look like this:

```
Option Explicit

Private Sub CloseButton_Click()
    Unload Me
End Sub

Private Sub DefaultButton_Click()
    Session.RestoreDefaults rcColors
End Sub

Private Sub RedButton_Click()
    Session.BackgndColor = rcRed
End Sub
```

ColorDemo Code for Reflection for HP, UNIX and OpenVMS, and ReGIS Graphics

If you are using Reflection for HP with NS/VT, Reflection for UNIX and OpenVMS, or Reflection for ReGIS Graphics, your UserForm code for the ColorDemo dialog box will look like this:

```
Option Explicit

Private Sub CloseButton_Click()


End Sub

Private Sub DefaultButton_Click()
    Session.RestoreDefaults rcColors
End Sub

Private Sub RedButton_Click()
    Session.SetColorMap rcPlainAttribute, rcWhite, rcRed
End Sub
```

Exercise 2: Getting User Input from a Dialog Box

This exercise demonstrates how to create a dialog box that changes in response to user input. The macro you create uses Reflection's **GetOpenFolder** method to allow a user to browse for files. The file the user selects is displayed in a text box. The finished dialog box is shown in the figure.

Creating a New UserForm

In steps 1 and 2, you open the Visual Basic Editor and create a new UserForm.

1. Open the Reflection settings file you are using for your practice macros. On the **Macro** menu, click **Macros.** In the **Macro name** box, type `OpenFileDemo`, then click **Create**.
2. Open the **Insert** menu in the Visual Basic Editor, and click **UserForm**.

Designing the Dialog Box

In steps 3-10, you modify the dialog box and add controls to it. For more detailed help on these techniques, see the ColorDemo macro exercise on page 52.

3. Use the Properties window to set the form **Name** to *OpenFileDialog* and **Caption** to *Select File*.
4. Use the sizing handles on the dialog box form to change its shape to match the completed dialog box shown above.
5. Add a Label control, a TextBox control, and three CommandButton controls. Arrange these controls as shown in the completed dialog box.
6. Set the Label **Caption** to *Path and file name*:
7. Set the TextBox **Name** to *SelectedPath*.
8. For the Browse button, set the **Name** to *BrowseButton* and **Caption** to *Browse*.
9. For the OK button, set the **Name** to *OKButton* and **Caption** to *OK*.
10. For the Cancel button, set the **Name** to *CancelButton* and **Caption** to *Cancel*.

Add Code to the Form

In steps 11-14, you add code to the form. The completed code for this dialog box is on page 61.

11. Double-click the Browse control to create an event procedure for this button and insert the code shown here. The Reflection **GetOpenFileName** method opens a standard Windows dialog box for selecting a file, and returns a string with the file name and path selected. In this procedure, the returned string is used to set the **Text** property of the SelectedPath text box. **Me** is used to identify the current UserForm; *Me.SelectedPath.Text* in this statement is equivalent to *OpenFileDemo.SelectedPath.Text*.

```
With Session
   Me.SelectedPath.Text = GetOpenFilename("All Files (*.*),*.*")
End With
```

12. Double-click the OK button, and insert the following statement in the event procedure. This statement closes the dialog box without removing it from memory.

```
Me.Hide
```

13. Double-click the Cancel button, and insert the following statement in the event procedure. This statement closes the dialog box and unloads it from memory.

```
Unload Me
```

14. To test the dialog box, return to the UserForm window (by clicking it or by using the Windows menu), then click the Run button on the Visual Basic toolbar. Click the Browse button in your dialog box. You should see the Open File dialog box created by **GetOpenFilename**. Select any file, then click Open. The file you selected should be displayed in the **Path and filename** text box.

Incorporating the Dialog Box into Your Macro

In steps 15 and 16, you add code to your macro that opens the dialog box and then shows a message box with the current value of the TextBox control. If you close the dialog box using the Close button, the message box displays the most recent contents of this control. If you close the dialog box using the Cancel button, the message box displays an empty string, because the OpenFileDemo dialog box is no longer loaded in memory.

15. Double-click NewMacros in the Project Explorer to open this module in the Code window.

16. Edit the OpenFileDemo procedure as follows:

```
Sub OpenFileDemo()
'
' OpenFileDemo Macro
' Macro created 09/30/98 by Your Name
'
   'Display the OpenFileDialog form
   OpenFileDialog.Show

   'Use a message box to display the contents of the text box
   MsgBox "User selected " & OpenFileDialog.SelectedPath.Text

End Sub
```

17. Test the macro and save your settings file.

TheOpenFileDemo Dialog Box UserForm Code

This is the completed code for the OpenFileDemo dialog box.

```
Option Explicit

Private Sub BrowseButton_Click()
With Session
   Me.SelectedPath.Text = GetOpenFilename("All Files (*.*),*.*")
End With
End Sub

Private Sub CancelButton_Click()
   Unload Me
End Sub

Private Sub OKButton_Click()
   Me.Hide
End Sub
```

CHAPTER 7

Handling Errors

There are the three types of errors you may encounter as you program in Reflection:

Compile errors
Compile errors prevent your macro from running and generally result from errors in syntax.

Programming logic errors
Logic errors occur when your macro does not perform as you expected. The programming syntax is correct, the macro compiles, but an error in logic causes the macro to produce unexpected or incorrect results. The Visual Basic Editor's debugging tools can help you track down logic errors. Search for *debugging code* in the Visual Basic Editor's Help index for more information.

Run-time errors
Runtime errors occur as your macro runs, and generally result from specific conditions present at that time. For example, a runtime error may occur if you prompt the user for a host name, try to connect to that host, but the host is not available. The **Connect** method fails and Visual Basic generates a runtime error.

You should always include some form of error handling in your macros to deal with runtime errors, even if you handle the error by doing nothing. Without any error handling, a runtime error causes a macro to stop immediately, and gives the user little information. This chapter covers the following topics:

- Trapping an error
- Handing the error
- Resuming the macro
- Inline error handling
- Error codes

Trapping an Error

The first step in dealing with runtime errors is to set a "trap" to catch the error. You do this by including an **On Error** statement in your macro. When a runtime error occurs, the **On Error** statement transfers control to an error-handling routine. Refer to the **On Error** statement topic in the Visual Basic Help for more information. (See page 2 for information about how to view this Help.)

To trap errors correctly, you must set your error trap above the point in the procedure where errors are likely to occur. In general, this means that your error trap should be placed near the top of the procedure. Further, to avoid having the error-handling routine execute even when no error occurs, include an **Exit Sub** or **Exit Function** statement just before the error-handling routine's label.

Examples

The following examples show the general structure of a procedure that includes an error trap and error-handling routine.

In the first example, the **On Error** statement specifies the name of an error handler (called MyHandler) to which control is transferred when a runtime error occurs. After the error is handled, the macro terminates:

```
Sub SampleWithoutResume ()
   On Error GoTo MyHandler
   ' Program code goes here. To avoid having the error handling
   ' routine invoked after this section is executed, include the
   ' following line just above the error-handler.
   Exit Sub

MyHandler:
   ' Error is handled and the macro terminates gracefully.
   Exit Sub

End Sub
```

The next example shows the general structure of a procedure in which control resumes at the line following the statement that caused the runtime error:

```
Sub SampleResumeNext ()
   On Error GoTo MyHandler
   ' Normal program code goes here. If an error occurs, control is
   ' transferred to the handler. When the handler is done, control
   ' resumes at the next line here.
   Exit Sub

MyHandler:
   ' Error is handled, and control resumes at the line after the
   ' statement that caused the error.
   Resume Next

End Sub
```

In the third example, after the error handler executes, control returns to the DoPrompt label.

```
Sub SampleResumeToLabel ()
   On Error GoTo MyHandler
DoPrompt:
   ' Normal program code goes here. If an error occurs, control
   ' is transferred to the handler.
   Exit Sub

MyHandler:
   ' Error is handled, then control is transferred to the top of
   ' the macro, at the DoPrompt label.
   Resume DoPrompt

End Sub
```

Error-Handling Routines

After setting an error trap, you must write the error-handling routine that deals with the errors that you anticipate arising when your macro runs. You can anticipate many of the common errors and specifically handle these cases. Include some code in your error-handling routine to catch errors you don't anticipate; this lets you deal with unanticipated errors that "fall through" your specific error-handling cases.

The error-handling routine specified by an error trap is identified by a line label. Typically, the error-handling routine is placed at the end of the procedure, just before the **End Sub** or **End Function** statement. Also, to avoid having the error-handling code executed even when no error occurs, an **Exit Sub** or **Exit Function** should be placed just before the error handler's label. A Resume statement is generally used to continue execution of the macro after the error handler is done.

Depending on how you want to deal with errors, you can write your error-handling routine in any number of ways. For example, a simple error handler that just displays a custom error message and then terminates the macro might look like this:

```
Sub ExitOnError ()
   On Error GoTo MyHandler
   ' Main body of procedure.
   Exit Sub

MyHandler:
   MsgBox "Error occurred. Cannot complete operation."
   Exit Sub

End Sub
```

A more complex error handler can use the **Err** object to return specific information about the error. (Refer to the Visual Basic Help for more information about the **Err** object.) For example, a "Path/File access error", with an error code of 75, is returned when you attempt to open a read-only file in sequential **Output** or **Append** mode. The following macro uses the **GetOpenfilename** method to open a File Open dialog box and request a file, then tries to write to that file. If the file is read-only, the error handler displays this information and returns the user to the File Open dialog box.

The line continuation character, an underscore preceded by a space, is used here to break up long lines of code.

```
Sub WriteToFile()
   Dim fileName As String
   On Error GoTo MyHandler
chooseAgain:
   fileName = GetOpenFilename("All Files (*.*),*.*", , _
           "File Open", "Open")
   Open fileName For Append As #1
   Write #1, "stuff"
   Close #1
   Exit Sub

MyHandler:
   'If the file is read-only, let the user try again.
   'The underscore character is used here to break the long line.
   If Err.Number = 75 Then
      MsgBox "Can't write to file" & fileName & _
         ". Please choose another.", , "File Error"
      Resume chooseAgain

   'For other errors display the error message, close the
   'open file and exit.

   Else
      MsgBox Err.Number & ": " & Err.Description, , "File Error"
      Close #1
      Exit Sub

   End If

End Sub
```

Note: If you have existing Reflection Basic error-handling code that was written without using the **Err** object, you do not need to rewrite it for use in Visual Basic. Visual Basic continues to support the **Error** function. Also, the default property of the **Err** object is **Number**. Because the default property can be represented by the object name **Err**, earlier code written using the **Err** function or **Err** statement doesn't have to be modified.

Resuming a Procedure After an Error

After you've successfully trapped and handled an error, you can either exit the procedure directly from the error handler by including an **Exit Sub** or **Exit Function** statement in the handler, or you can continue execution of the procedure. Before resuming the procedure, you may even be able to correct the error condition automatically, in which case the user may never know that an error occurred.

To resume a macro from an error-handling routine, use the **Resume** statement. There are three forms of the **Resume** statement:

Resume
The **Resume** statement by itself exits the error handler and resumes the macro at the line that caused the error. Of course, your macro must automatically correct the error condition or prompt the user to correct it before resuming; otherwise, the error will occur again.

Resume Next
This form of the **Resume** statement resumes the macro at the line following the one that caused the error.

Resume Label
This form of the **Resume** statement exits the error handler and passes control to the statement identified by the label.

Inline Error Handling

If you want a procedure to handle relatively simple runtime errors without having to branch to a separate error-handling routine, you can use a form of the **On Error** statement that lets you deal with errors "in line"; that is, directly in the code that caused the error, rather than in a separate routine.

To handle an error in line, use **On Error Resume Next**. With this form of the **On Error** statement, any errors that occur during run time simply cause Reflection to continue executing the macro at the next statement. The lines following the error should then determine if an error occurred and handle the error by opening a dialog box, passing control to another procedure or to a routine within the same procedure.

Example

This example checks for errors when saving a Reflection for UNIX and OpenVMS settings file. (To see examples for other Reflection products, open the Reflection Programming Help and search for *Error handling in macros, inline error handling*. See page 2 for information about viewing this Help.) If the save operation fails, an error message is displayed. If the save operation succeeds (and therefore **Err.Number** returns 0), a message to indicate success is displayed. The inline error handler is needed because without it, the macro would simply terminate with an error if the **SaveSettings** method failed.

```
Sub SaveSettingsDemo()
   Dim theError As Integer

   On Error Resume Next
   Session.SaveSettings "Settings.r2w", rcSettings

   theError = Err.Number

   Select Case theError
      Case 0
         MsgBox "Save complete."
      Case Else
         MsgBox Err.Description & "."
   End Select

End Sub
```

Information About Error Codes

Error codes between 1 and 1000 are returned if an error occurs while executing a Visual Basic command. For more information about these errors, see *trappable errors* in the Visual Basic Help. Reflection-specific error codes have different values depending on the Reflection product you are using.

Reflection for IBM

Error codes between 4000 and 4999 are returned if an error occurs while executing a command supported by Reflection for IBM. To see a list of these errors and their corresponding error messages, search for *Error codes, error codes and constants* in the Reflection Programming Help. These errors can also be represented by error constants beginning with "rcRte". For example, the predefined constant rcRtePathNotFound is equivalent to 4023. You can use predefined constants in your macros to help make them more readable.

Reflection for HP, Reflection for UNIX and OpenVMS, and Reflection for ReGIS Graphics

Error codes between 10,000-10,999 are returned if an error occurs while executing a command supported by these Reflection products: Reflection for HP, Reflection for UNIX and OpenVMS, Reflection for ReGIS Graphics. To see a list of these errors and their corresponding error messages, search for *Error codes, error codes and constants* in the Reflection Programming Help. These errors can also be represented by error constants beginning with "rcErr". For example, the predefined constant rcErrNot-Connected is equivalent to 10,023. You can use predefined constants in your macros to help make them more readable.

CHAPTER 8

Communicating with Other Applications

Visual Basic uses a standard set of protocols called Automation (or OLE Automation) to allow one application to communicate with other applications. This chapter covers the following Automation topics:

- Understanding Automation
- Controlling other applications from Reflection
- Controlling Reflection from other applications
- Using **CreateObject** and **GetObject**
- Using Reflection predefined constants in other applications

Understanding Automation

Any application that supports Automation can communicate with any other. This means that a Reflection session can communicate with other Reflection sessions, Microsoft Office products, Visio, stand-alone Visual Basic, or any other product that supports Automation. Automation support provides a standardized way to:

- Control other applications from Reflection. For example, from Reflection, you can start Excel, copy data from a host screen to an Excel spreadsheet, save the spreadsheet, and exit Excel. In this situation, Reflection is the controller (or client) that manipulates Excel, which is the object (or server).
- Control Reflection from other applications. For example, from Word you can launch a Reflection session, log onto a host, then copy screen data to the current document. In this situation, Reflection is the object (or server) that is being manipulated by Word, which is the controller (or client).

Reflection's Automation support is provided by Visual Basic for Applications and Reflection's methods and properties. When you want to manipulate another application, use Visual Basic commands to create an object for that application, then control that object using its methods and properties. When you want to communicate with Reflection from another application, create a Reflection object and use Reflection's methods and properties in the other application's programming environment to extract data from or send instructions to Reflection.

Controlling Other Applications from Reflection

Use these steps when you want to use Reflection to control an application that supports Automation (such as Word or Excel):

1. Add a reference to the object library for the application you want to control. To do this, click References on the Visual Basic Editor's Tools menu, select the application you want from the **Available References** list, then click OK.
2. In your procedure code, use **Dim** to dimension an object variable for the object you want to control. For example, this statement dimensions an Excel object:

```
Dim excelApp As Excel.Application
```

3. Use **Set** to assign an object to the object variable. Use either **CreateObject** or **GetObject** to identify the object. For example, to create a new instance of Excel:

```
Set excelApp = CreateObject("Excel.Application")
```

4. Use the object to manipulate the application by using commands supported by that application. For example, these statements use Excel objects, methods, and properties to make Excel visible, create a new Excel workbook, and put the number 14 in cell B2:

```
excelApp.Visible = True
excelApp.Workbooks.Add
excelApp.ActiveSheet.Range("B2").Select
excelApp.ActiveCell.Value = 14
```

Example

This procedure copies text from the Reflection screen display and enters it into a Word document. The line continuation character (an underscore preceded by a space) is used here to break up long lines.

Note: Use **GetText** to get display text in Reflection for HP, UNIX and OpenVMS, and ReGIS Graphics. Use **GetDisplayText** in Reflection for IBM. In the example, the command for Reflection for IBM is commented out.

```
Sub SendDisplayInfoToWord()
   Dim displayText As String

   With Session
      'Get text from the screen display

      'For Reflection for HP, UNIX and OpenVMS, RegGis Graphics:
      displayText = .GetText(.ScreenTopRow, 0, _
              .ScreenTopRow + .DisplayColumns, .DisplayColumns)

      'For Reflection for IBM:
      'displayText = .GetDisplayText(1, 1, _
      '       .DisplayColumns * .DisplayRows)

   End With

   'Create a Word object
   Dim Word as Word.Application
   Set Word = CreateObject("Word.Application")

   'Make Word visible and create a new document
   Word.Visible = True
   Word.Documents.Add

   'Add the display text to the document
   Word.Selection.TypeText Text:=displayText

   'Save the document and quit Word
   Word.ActiveDocument.SaveAs Filename:="C:\MySample.doc"
   Word.Quit

End Sub
```

Controlling Reflection from Other Applications

Use the following guidelines for controlling Reflection from stand-alone Visual Basic or from other applications (such as Word and Excel) that use Automation:

1. In the procedure you are writing in the other application, dimension an object variable for the Reflection object. For late binding, use the following statement. (Refer to the Visual Basic Help for **CreateObject** for more information about early and late binding.)

   ```
   Dim MyReflectionObject As Object
   ```

   For early binding, use one of the following statements, depending on the Reflection product you are using:

   ```
   'Reflection for IBM
   Dim MyObject As Reflection.Session

   'Reflection for HP
   Dim MyObject As Reflection1.Session

   'Reflection for UNIX and OpenVMS
   Dim MyObject As Reflection2.Session

   'Reflection for ReGIS Graphics
   Dim MyObject As Reflection4.Session
   ```

2. Use **Set** to assign a Reflection object to the object variable. You can create a new instance of Reflection at this time, or attach to an existing Reflection object.

   - Use **CreateObject** to create a new Reflection object. (See "Using CreateObject" on page 76 for more information.)
   - Use **GetObject** to attach to an existing Reflection object. (See "Using GetObject" on page 77 for more information.)

   For example, this statement creates a new Reflection for IBM object.

   ```
   Set MyReflectionObject = CreateObject("ReflectionIBM.Session")
   ```

Refer to the Visual Basic Help for additional information about **Set**, **CreateObject**, and **GetObject**.

3. Use the Reflection object you just created to access Reflection's methods and properties. The **CreateObject** function launches Reflection but does not make the application visible. If you want Reflection to be visible while the other application uses it, use Reflection's **Visible** property. For example:

   ```
   MyReflectionObject.Visible = True
   ```

Using CreateObject

Use **CreateObject** to create a new instance of an Automation application. **CreateObject** takes one argument of the form **AppName.ObjectType**. AppName is the name of an application and ObjectType specifies the object to create. (Many applications support several objects. Reflection has only one object, and all Reflection methods and properties act on this object.)

The following examples show how to use **CreateObject** to create a Reflection object for different Reflection products:

To create a Reflection for IBM object:

```
Dim ReflectionIBM As Reflection.Session
Set ReflectionIBM = CreateObject("ReflectionIBM.Session")
```

To create a Reflection for HP object:

```
Dim ReflectionHP As Reflection1.Session
Set ReflectionHP = CreateObject("Reflection1.Session")
```

To create a Reflection for UNIX and OpenVMS object:

```
Dim ReflectionUO As Reflection2.Session
Set ReflectionUO = CreateObject("Reflection2.Session")
```

To create a Reflection for ReGIS Graphics object:

```
Dim ReflectionGraphics As Reflection4.Session
Set ReflectionGraphics = CreateObject("Reflection4.Session")
```

Using GetObject

GetObject returns a reference to an object. With most applications, you can specify a file name to identify the object. For example, you can access an open Word document like this:

```
Dim Word As Object
Set Word = GetObject("C:\Mypath\Mydoc.doc")
```

To use **GetObject** to access a Reflection session, you can use two different strategies. **GetObject** supports two arguments; the first argument, pathname, specifies a path and file name (for most applications); the second specifies an application name and object type. The following examples show how to use each of these arguments to attach to a Reflection session:

Attaching to Reflection Using the OLE Server Name

This technique uses the first argument to the **GetObject** function. The value you use for this argument should be the OLE server name for that session. This name is specified in Reflection using the **OLEServerName** property. Because you can specify a unique OLE server name for every Reflection session, this technique allows you to identify a particular Reflection session even if multiple sessions are active. The default values for this property are:

Reflection for HP: “R1WIN”
Reflection for UNIX and OpenVMS: “R2WIN”
Reflection for ReGIS Graphics: “R4WIN”
Reflection for IBM: “RIBM”

The following example shows how to attach to an instance of Reflection for ReGIS Graphics using the default **OLEServerName** value:

```
Dim ReflectionReGIS As Object
Set ReflectionReGIS = GetObject("R4WIN")
```

Attaching to Reflection Using the Session Object

This technique uses the second argument to the **GetObject** function, which has the format *AppName.ObjectType*. The application name for Reflection products are:

Reflection for HP: “Reflection1”
Reflection for UNIX and OpenVMS: “Reflection2”
Reflection for ReGIS Graphics: “Reflection4”
Reflection for IBM: “ReflectionIBM”

All Reflection products support a single object called **Session**.

The following example shows how to attach to Reflection for IBM. The comma is needed to indicate that the first argument is being omitted.

```
Dim RibmObject As Object
Set RibmOjbect = GetObject(, "ReflectionIBM.Session")
```

This technique works well if there is only one session running for any given Reflection product. If there are multiple Reflection sessions running, using this technique makes the attachment to an arbitrary instance of Reflection.

Note: If you include an empty string for the first argument, **GetObject** will open the specified application.

For example, these commands create a new instance of Reflection for HP:

```
Dim ReflectionHP As Object
Set ReflectionHP = GetObject("", "Reflection1.Session")
```

Using Reflection Predefined Constants in Other Applications

Reflection uses many predefined constants; they are used as method arguments, property values, and error codes. Using predefined constants rather than numbers makes macros easier to read. The examples below show pairs of equivalent statements. In each pair, the first statement uses a Reflection predefined constant, and the second statement uses the numeric equivalent.

Equivalent ways to save a settings file in Reflection for UNIX and OpenVMS:

```
Session.SaveSettings "Myfile.r2w", rcSettings, rcOverwrite
Session.SaveSettings "Myfile.r2w", 1, 1
```

Equivalent ways to set the value of the **GraphicsPrintMode** property in Reflection for ReGIS Graphics:

```
Session.GraphicsPrintMode = rcRotate
Session.GraphicsPrintMode = 2
```

Equivalent ways to handle an error in Reflection for IBM:

```
If Err.Number = rcRteNoFileAccess Then
If Err.Number = 4025 Then
```

If you are programming using stand-alone Visual Basic or you are using VBA in another application, you can use Reflection's predefined constants by adding a reference to the Reflection object library. To do this, click **References** on Visual Basic's **Tool** menu, and select the object library for the Reflection product with which you'll be communicating.

If you want to incorporate Reflection's predefined constants in other programming environments, use one of the following files (depending on the Reflection product you are using). These files contain symbolic names and values for all the constants used by Reflection.

- Rodecls.bas defines constants for Reflection for IBM. Look for this file in \Program Files\Attachmate\Reflection\Ibm\Samples\Vb.
- Rwinapi.txt defines constants for Reflection for HP, Reflection for UNIX and Digital, and Reflection for ReGIS Graphics. You can download this file from our web site. Refer to tehchnical note 1536 (available at http://support.wrq.com/techdocs/1536.html) for more information about downloading sample files.

CHAPTER 9

Using Events

Reflection provides the following ways to programmatically manage what happens in response to events that occur during a Reflection session:

- Automation events
- Methods and properties for managing events

This chapter provides an overview of these event types. To see a complete list of all of Reflection's event management commands, see Reflection's Programming Help.

Automation Events

Automation events are a standard part of many application programming interfaces. Events of this type are part of an application's object library and are identified in the Visual Basic Editor Object Browser with a lighting bolt icon. Using Automation events for event management in Reflection has the following advantages:

- With Automation events, you can create procedures that are triggered whenever the specified events occur. No macro needs to be running at the time the events occur.
- Automation events provide the only way to interrupt a process that has already been initiated. This capability is provided by any event that begins with the word "Before." For example, the **BeforeTransmitString** event is triggered before Reflection sends text to the host. You can create a **BeforeTransmitString** event handler to check and edit text before it is sent to the host.
- Any application or tool that can respond to Automation events (for example, another Reflection session, Microsoft Office applications, Visual Basic, PowerBuilder, C++, and so on) is capable of handling Automation events. This capability provides much tighter integration by allowing two-way communications between Reflection and another application.

Creating an Event Procedure

Automation event procedures must be created in the Reflection Session object (**ThisSession**). Use this procedure:

1. Open the Visual Basic Editor. (On Reflection's **Macro** menu, click **Visual Basic Editor**). If the Project Explorer is not visible, open the Visual Basic Editor's **View** menu and click **Project Explorer**.
2. Find the Reflection session object (**ThisSession**) in the Reflection objects folder. Double-click **ThisSession** to open a Code window for this object.
3. Select **Session** using the **Object** list box in the upper-left corner of the code window.

   Note: This will automatically insert an event procedure into the code window. You can delete this if you are not creating this kind of procedure.
4. Use the **Procedures/Events** list box in the upper-right corner of the code window to select the kind of event procedure you want to create. This will insert an empty event procedure of the type you selected.
5. In the procedure you just created, enter the code you want executed when the event occurs. For example, the following code uses the **BeforeExit** event to display a message box that asks the user to confirm an exit command. If the user clicks No, the exit is cancelled.

```
Private Sub Session_BeforeExit(Continue As Boolean)
   If MsgBox("Exit Reflection?", vbYesNo) = vbNo Then
      Continue = False
   End If
End Sub
```

Automation events are saved as part of your VBA project when you save your settings file.

Methods and Properties for Managing Events

Reflection also provides methods and properties that you can use for event management. Use these commands if:

- You want to programmatically create events that will show up in the Reflection Events Setup dialog box.
- You want to define a unique event, such as the cursor moving to a specified row and column.

Methods and properties for managing events fall two groups: **OnEvent** and **DefineEvent**. A general description of each group is provided here. To see a list of the commands for each group, see Reflection's Programming Help.

OnEvent Methods and Properties

Commands in this group are equivalent to using Reflection's Event Setup dialog box. You can add items to Reflection's Event Setup dialog box using the **OnEvent** method. Changes you make to Reflection using these commands can be saved to a settings file.

DefineEvent Methods and Properties

Commands in this group are defined using the **DefineEvent** method. This method defines an event without associating a specific action with that event. You can use subsequent commands to check for the occurrence of a defined event, or to wait for a defined event to happen. Commands in this group have no effect on the Event Setup dialog box. Events you define using the methods in this group are not saved as part of your settings file; they must be created by running the macro that contains the command.

CHAPTER 10

Managing Connections to IBM Hosts

This chapter describes techniques for making connections with Reflection for IBM. The following topics are covered:

- Commands for connecting and disconnecting
- Using connect macros
- Configuring connection settings

Commands for Connecting and Disconnecting

To establish a connection in a macro, use the **Connect** method. If a connection is already active when you issue the **Connect** method, this method is ignored.

You can check for an open connection by using the **Connected** property.

To disconnect from the host, use the **Disconnect** method.

Using Connect Macros

A connect macro is one that runs automatically when Reflection successfully makes a host connection. This procedure uses the macro recorder to create a connect macro:

1. (Omit this step if your settings are already configured to connect to your host.) On the **Connection** menu, click **Session Setup**. Use the Session Setup dialog box to configure your host connection. Don't connect yet. Click OK to close the dialog box.
2. Click **Start Recording** on the **Macro** menu.

3. Connect to the host using the **Connect** command on the **Connection** menu, then log on as you usually do.

   Note: Reflection will also connect to your host if you press the Enter key when you are disconnected. Don't use this shortcut when you are recording connections because this keystroke is also recorded.

4. Click **Stop Recording**. In the Stop Recording dialog box, enter a name for your macro in the **Macro name** box and enter an optional **Description**. Select the **Make this the connect macro** check box. Click **OK** to save the macro.

   Macro names cannot include spaces and must begin with a letter. See page 38 for more information about naming macros.

5. At this point you can run your macro, but it is not yet saved. Click **Save** on the **File** menu to save your settings and macros.

Note: Connect macros run as soon as a host connection is successfully established. A connect macro can help simplify host log-on procedures and automate the process of navigating to a particular host screen. Because you cannot change your session configuration while you are connected, connect macros should not include session setup commands.

If you have an existing macro that you want to designate as the connect macro, use either of these techniques:

- In Reflection, click **Session Setup** on the **Connection** menu. Enter the name of the macro in the **Connect macro** box (or use the Browse button to select a macro).
- In a macro, use the **ConnectMacro** property, for example:

```
Session.ConnectMacro = "LogonToMyHost"
```

You can pass additional information to a connect macro using either of these techniques:

- In Reflection, click **Session Setup** on the **Connection** menu. Enter the information in the **Macro data** box.
- In a macro, use the **ConnectMacroData** property, for example:

```
Session.ConnectMacroData = "one two three"
```

Note: You can also automate connection events using Reflection's Events Setup dialog box. One of the available events is *When a connection is made*. Click the Help button in the Events Setup dialog box for more information.

Configuring Connection Settings

Because macros and connection settings are saved in the same settings file, you do not need to change connection settings programmatically if you have already saved correct connection information in the settings file that contains your macros.

You may want to configure connections programmatically if:

- You are sharing macros with someone who has added a reference to a settings file containing your macros.
- You are using Automation to create new Reflection sessions or to attach to existing sessions.
- You are creating several macros that connect to different hosts using different connection settings.

Using the Macro Recorder to Configure Connections

A good strategy for managing connections in macros is to use the macro recorder to capture connection information. You can begin with an untitled Reflection session, record a connection, then copy the recorded information to the macro you are developing. To do this:

1. Open the settings file that contains your macros.
2. On the **File** menu, click **New Session** to open an untitled Reflection for IBM session.
3. Start the macro recorder in the new session.
4. Use the Session Setup dialog box in the new, untitled session to configure a connection to your host, then click either **OK** or **Connect** to close this dialog box. (If you click **OK**, the recorded code will include session configuration information, but will not include making the connection.)
5. Stop the macro recorder. By default, the recorded macro is placed in your new, untitled Reflection session. The remaining steps describe how to copy this code to your original settings file.
6. In the Stop Recording dialog box, change **Destination** to **Clipboard**. When you make this change, a **Clipboard options** area appears in the dialog box. In this area, set **Syntax** to **VBA Source** and clear the **Include object prologue** check box. Click **OK**.
7. Return to your original Reflection session, open the Visual Basic Editor, display the code module you want to contain your code, and use the Paste command to paste your recorded code.

Note: Do not use this procedure to create a connect macro. Connect macros run after a connection has already been established and should not include session setup information.

Reflection Methods and Properties for Establishing and Managing Connections

Use the **SetupSession** method to configure a connection. With **SetupSession**, you specify a SessionType (what type of host terminal or printer Reflection is to emulate), a TerminalModel (the specific host terminal or printer Reflection is to emulate), and a TransportType (what data communications software Reflection uses to communicate with the host). An error results if you try to use **SetupSession** when you already have a host connection.

Several read-only properties let you find out about the current connection: The **SessionType** property returns the type of host terminal or printer Reflection is emulating. The **TerminalModel** property returns the specific host terminal or printer Reflection is emulating. Both **TransportType** and **TransportName** return the transport Reflection is using to connect to the host, **TransportType** returns a numeric value indicating the transport type, and **TransportName** returns a string. When no connection exists, these properties return values indicating which connection settings Reflection would use if a connection was established.

The Reflection Programming Help includes lists of the properties you can use to configure connections for specific hosts and transports. Search for *Connection keywords (programming)* in the index.

Note: For 802.2 , Coax DFT, and SDLC, and connections, links configuration is handled by the Reflection SNA engine. This information is stored in the SNA Engine configuration file. By default, this file is called Wrqsna.rlf. You can specify a new name and location for this file with the **SNAEngineCFGFile** property.

CHAPTER 11

Managing Connections to HP, UNIX, OpenVMS, and Unisys Hosts

This chapter describes techniques for making connections with the following Reflection applications:

Reflection for HP
Reflection for UNIX and OpenVMS
Reflection for ReGIS Graphics

These topics are covered:

- Commands for connecting and disconnecting
- Using connect macros
- Configuring connection settings
- Managing modem connections
- Handling connection errors

Commands for Connecting and Disconnecting

To establish a connection in a macro, use the **Connect** method. If a connection is already active when you issue the **Connect** method, a runtime error occurs. You can check for an open connection by using the **Connected** property.

To disconnect from the host, use the **Disconnect** method. Unlike the **Connect** method, which must always be used to open a connection, **Disconnect** may not be necessary. When you log out of the host computer, the network connection is typically closed automatically. You can use the **Disconnect** method, however, to ensure that the connection is closed. If the connection is already closed, the **Disconnect** method generates an error. You should trap this error with an error-handling routine.

For serial connections (either direct serial or modem connections), the **Disconnect** method closes the serial port. You should always use the **Disconnect** method to ensure that the serial port is closed when you're done with it; this makes the port available to another session or application.

By default, when you close a serial connection (either direct or modem)—no matter whether you close it with the **Disconnect** method, the Disconnect command on the Connection menu, or by quitting Reflection—the DTR (data terminal ready) signal is also dropped. In some cases, particularly with modem connections, you may want the DTR signal to remain true even after disconnecting or quitting Reflection; this prevents the modem from hanging up and lets you return to your Reflection session and resume your connection. To avoid dropping the DTR signal when disconnecting, set the **ConnectionSettings** keyword DropDTROnDisconnect to **False**. More information about this keyword is available in the Reflection Programming Help. (See page 2 for instructions on how to view this Help.)

Example

In the following example, a connection is configured and the **Connect** method is used to open the connection before waiting for a host prompt. If the host prompt is not received after 10 seconds, the connection is closed.

```
Sub MakeMyConnection ()

Dim isFound As Integer
Dim timeOut As Integer
timeOut = 10

With Session
   If .Connected = False Then
      .ConnectionType = "VT-MGR"
      .ConnectionSettings = "Host MyHPHost"
      .Connect
      isFound = .WaitForString("MPE XL:", timeOut, rcAllowKeystrokes)
      If isFound = False Then
         .Disconnect
      End If
   Else
      MsgBox "There is already a connection open."
   End If
End With
End Sub
```

Using Connect Macros

A connect macro is one that runs automatically when Reflection successfully makes a host connection. You can specify a connect macro using any of these techniques:

- When you record a login macro, select the **Make this the connect macro** check box in the Stop Recording dialog box.
- In Reflection, click **Connection Setup** on the **Connection** menu, and click the **Connect Macro** button. This opens the Connect Macro dialog box. Enter the macro name in the **Macro name** box (or use Browse to select a macro).
- In a macro, set the ConnectMacro keyword token of the **ConnectionSettings** property, for example:

```
Session.ConnectionSettings = "ConnectMacro Login"
```

You can pass information to a connect macro using either of these techniques. (To return this value in the macro, use the **MacroData** property.)

- In Reflection, click **Connection Setup** on the **Connection** menu, and click the **Connect Macro** button. This opens the Connect Macro dialog box. Enter the information you want to pass to the macro in the **Macro data** box.
- In a macro, use the ConnectMacroData keyword token of the **ConnectionSettings** property, for example:

```
Session.ConnectionSettings = "ConnectMacroData ""one two
three"""
```

If you have upgraded from an earlier version of Reflection and have already designated a Reflection Basic connect script, Reflection will continue to use your script as the connect script.

Note: You can also automate connection events using Reflection's Events Setup dialog box. One of the available events is *When a connection is made*. Click the Help button in the Events Setup dialog box for more information.

Configuring Connection Settings

Macros and connection settings are saved to the same settings file, so you do not need to change connection settings programmatically if you have already saved correct connection information to the settings file that contains your macros.

You may want to configure connections programmatically if:

- You are sharing macros with someone who has added a reference to a settings file containing your macros.
- You are using Automation to create new Reflection sessions.
- You are creating several macros that connect to different hosts using different connection settings.

Determining the Current Connection Type

The **ConnectionType** property is used both to find out the current connection type and to specify a connection type.

Because each type of connection has its own set of configuration options, if you're writing a macro that manipulates the current connection settings, you may want to determine the current settings before changing them. After determining the current connection type, you can use the **ConnectionSetting** method to retrieve the value of a single connection parameter, or the **ConnectionSettings** property to retrieve or set multiple connection parameters.

Example

In the following example, the connection type is retrieved and displayed in a message box. If there is no connection currently configured (the **ConnectionType** is **NONE**), a different message is displayed:

```
Sub DisplayCurrentConnection ()
   Dim how As String

   how = Session.ConnectionType
   If how = "NONE" Then
      how = "There is no connection currently configured."
   Else
      how = "The current Connection Type is " & how
   End If
   MsgBox(how)
End Sub
```

Specifying a Connection Type

If you want to change connection settings in a macro, you use **ConnectionType** property both to determine the current connection type and to specify a new connection type.

If you want Reflection to attempt the connection using its **Best Network** option, specify a **ConnectionType** of **BEST-NETWORK**. After the Best Network connection is established, the DefaultNetwork keyword of the **ConnectionSettings** property is set to the actual network connection type that was used to establish the connection.

When you change the connection type using the **ConnectionType** property, all connection settings for that type (that is, all settings you can configure with the **ConnectionSettings** property) are reset to their default values.

If a connection is already active when you specify a connection type, a runtime error occurs. Use the **Connected** property, as shown in the example, to determine if a connection is already active before trying to set the **ConnectionType** property.

Example

The following example first determines whether a connection is open. If not, the example configures a Telnet connection, specifies a host, and opens the connection. If the connection is already open, a message box reports the current connection to the user.

```
Sub ConnectToMyHost ()
   Dim how As String

   With Session
      If .Connected = False Then
         .ConnectionType = "TELNET"
         .ConnectionSettings = "Host myHost"
         .Connect
      Else
         how = "You are currently connected using " & _
              .ConnectionType
         MsgBox how
      End If
   End With
End Sub
```

Configuring Settings for a Connection Type

Once you have specified a connection type with the **ConnectionType** property, use the **ConnectionSettings** property to configure the individual options for the connection type. (The **ConnectionSettings** property can also be used to determine the current settings for a connection type.)

The **ConnectionSettings** property has the following syntax:

```
Session.ConnectionSettings = StringValue
```

The StringValue consists of alternating keyword tokens and value tokens. The keyword token specifies a setting; the value token specifies a value for the setting. For example, in the statement:

```
Session.ConnectionSettings = "Parity 8/None"
```

the keyword *Parity* is given a value of *8/None*.

Different connection types have different keyword/value pairs, and both the keywords and values are specific to the connection type (though many of the keywords are valid for a number of different connection types).

The entire configuration string must be enclosed in quotation marks. If a value token contains double quotation marks or a backslash character, you must precede the character with a backslash character. If a value token contains spaces, the token must be enclosed in single quotes or in two sets of quotation marks; for example:

```
Session.ConnectionSettings = "ConnectMacroData 'a b c'"
```

or

```
Session.ConnectionSettings = "ConnectMacroData ""a b c"" "
```

For any given connection type, the complete **ConnectionSettings** string can be quite lengthy. For example, the connection settings string for a Telnet connection might look like this:

```
CheckParity False Parity 8/NONE CharDelay 0 Host "" TelnetPort 23 Telnet-
Break False TelnetBinary False TelnetLFAfterCR False
TelnetInitOptionNegotiation True TelnetTermType VT220 SettingsFile ""
TelnetEcho Auto TelnetUseEmulatorTermType False TelnetSetWindowSize True
TelnetLinemode Never UseSOCKS False TelnetTrace False ExitAllowed True
ExitOnDisconnect False UseThreadedIO True ConnectMacro "" ConnectMac-
roData "" ConnectionName ""
```

Although you could specify an entire string like this as the **ConnectionSettings** property, you typically do not need to specify everything to establish a connection. Instead, you can issue an abbreviated configuration string containing only the keyword and value tokens for the settings you need to change; current values are used for the keywords you do not specify. For example, if you want to change only the setting for allowing an exit while connected, you could use this statement:

```
Session.ConnectionSettings = "ExitAllowed False"
```

All other settings for the current connection type remain unchanged.

If the string you assign to the **ConnectionSettings** property contains any invalid keyword or value tokens (for example, a token is not valid for the current connection type), a runtime error occurs. An error also results if a connection is currently active and you try to set a keyword that cannot be changed while the connection is open. Your macro should contain an error-handling routine to trap and deal with these errors (by displaying a message box, for example); you can retrieve the error message text with the **ConnectionErrorMessage** property.

Example

In the following example, a Telnet connection is configured, a host name is specified using an abbreviated configuration string, and a connection is opened. An input box is used to prompt for the host name. Error handling has been omitted from this example for simplicity.

```
Sub ConnectToTelnetHost ()
   Dim whatHost As String
   With Session
      If .Connected = True Then
         MsgBox "You already have an open connection."
      Else
         .ConnectionType = "TELNET"
         whatHost = InputBox("Host to connect to:", "Host Name")
         If whatHost <> "" Then
            .ConnectionSettings = "Host " & whatHost
            .Connect
         End If
      End If
   End With
End Sub
```

Determining the Settings for a Connection Type

In macros, you use the **ConnectionSettings** property to configure the settings for a selected connection type. The **ConnectionSettings** configuration string for any given connection type can consist of just a few keyword tokens, or it can consist of more than a dozen keyword tokens. Further, each keyword token can take many different value tokens.

Determining the correct keyword and value tokens for the **ConnectionSettings** string can be difficult, and determining the correct configuration string can be tedious. Besides, to establish a host connection, you typically do not need to configure every setting for a connection type, but rather just one or two. "Configuring Settings for a Connection Type" on page 98 explains how to use the **ConnectionSettings** property to configure a connection type using either a complete configuration string or an abbreviated configuration string.

There are three ways to determine the keyword and value tokens you should use for a given connection type:

- Capture the connection settings with the Reflection macro recorder. This is the easiest and recommended method. (See below.)
- Consult the tables that list all of the valid keyword and value tokens for the connection types that Reflection supports. To display these tables, open the Reflection Programming Help, and search for *Keyword/Value tokens*. (See page 2 for instructions on how to view this Help.)
- Use values returned by the **ConnectionSettings** property or the **ConnectionSetting** method.

Capturing Connection Settings Using the Reflection Macro Recorder

When you are writing macros with the **ConnectionSettings** property, the easiest way to determine which keywords to change, what values to change them to, and the correct configuration string to issue, is to use the Reflection macro recorder. With the macro recorder, you use Reflection dialog boxes to configure the connection options you want; the macro recorder captures only the parameters you change from their defaults, making the resulting macros very efficient and concise.

The following steps describe how to use the macro recorder to capture changes you make to connection settings after starting with the default settings for a connection type. You will open a new Reflection session, record a connection, and copy the resulting macro code to the Clipboard. You can then return to the Reflection session that contains your macros and paste the code into a code module.

1. Open the settings file that contains your macros.
2. On the **File** menu, click **New Session** to open an untitled Reflection session.
3. Start the macro recorder in the new, untitled session.
4. Use the Connection Setup dialog box in the untitled session to configure a connection to your host, then click Connect.

   Note: The macro recorder will not capture your connection settings unless you successfully complete a connection to the host.
5. Stop the macro recorder. By default, the recorded macro would be placed in your new, untitled Reflection session. The remaining steps describe how to copy this code to your original settings file instead.
6. In the Stop Recording dialog box, change **Destination** to **Clipboard**. In the **Clipboard Options** area, set **Format** to **Reflection macro**. Clear the **Include Object prologue** check box (unless you want to create a new procedure using this code.)

7. Return to your original Reflection session, open the Visual Basic Editor, display the code module you want to contain your code, and use the Paste command to paste in your recorded code.

When you capture connection settings using this technique, the correct syntax, including spaces and quotation marks, is automatically recorded. In most cases, you can simply copy and paste the recorded commands into your own procedures; modification of the **ConnectionSettings** lines is rarely needed. You may want to add your own error-handing routines, however, to take into account times when the host is unavailable, or the connection fails for some other reason.

Managing Modem Connections

By default, Reflection makes modem connections using the modem you have configured with Windows Control Panel.

To make a modem connection:

1. Set the **ConnectionType** property to "MODEM".
2. Use the **ConnectionSettings** property to specify the modem parameters you need. Use the Reflection Programming Help to view a list of keyword tokens you can use to specify modem settings. Look up *Modem, Tables of Keyword/Value tokens*. (See page 2 for instructions on how to view this Help.)
3. Use the **Connect** method to open the connection.

   Note: If no modem has been configured, a dialog box opens asking you if you want to run the Modem Control Panel to add a modem driver. If you select Yes in this dialog box, the Install New Modem wizard starts and Reflection opens a second dialog box instructing you to click OK when modem installation is complete. When you close this dialog box, Reflection dials the newly installed modem. If you do not run the Install New Modem program or cancel the installation process before modem installation is complete, **Connect** will return rcGeneral-ConnectionError with **ConnectionErrorMessage** set to *No modems installed*.
4. At the end of the modem session, use the **Disconnect** method to hang up the modem.

Example

The following example configures a connection. The area code and country are used to determine appropriate dialing prefixes. (The line continuation character, an underscore preceded by a space, is used here to break up a long line of code.)

```
Sub WRQDemo ()
With Session
   .ConnectionType = "MODEM"
   .ConnectionSettings = "ModemUseDialingRules True"
   .ConnectionSettings = "ModemPhoneNumber 999-1234"
   .ConnectionSettings = "ModemAreaCode 555"
   .ConnectionSettings = _
          "ModemCountryName 'United States of America (1)'"
   .ConnectionSettings = "ModemLocation 'Default Location'"
   .Connect
End With
End Sub
```

Note: If you have used versions of Reflection prior to version 6.0, you may have developed Reflection Basic scripts that use Reflection's modem dialer rather than the Control Panel Modem. To configure Reflection to use the old modem dialer, set the **UseModemDialerV5** property to True. If you are using the old modem dialer, you can use the **Dial** method to invoke the modem dialer, which initializes the modem and dials. The **Dial** method supports a number of arguments to specify a phone number, an initialization string to send to the modem, and others.

Handling Connection Errors

When you write a macro that makes a connection, you should write an error handling routine to trap any runtime errors that may occur when Reflection attempts the connection. With an error handler, you can provide additional feedback to the user, if necessary, and proceed with the macro as appropriate; without an error handler, the macro stops at the point where the error occurred, and Reflection displays an error message. See Chapter 6 (page 63) for more information about error handling.

If a connection-related error occurs, you can trap the error with the **On Error** statement and use the **Number** property of the **Err** object (**Err.Number**) to determine which error occurred. Once you have trapped the error, you can retrieve a predefined text string that describes the error, using either **Err.Description** or the **ConnectionErrorMessage** property. Then, you can have your macro proceed as needed.

Note: If you have existing error-handling code that was written without using the **Err** object, you do not need to rewrite it for use in Visual Basic. Visual Basic continues to support the **Error** function. Also, the default property of the **Err** object is **Number**. Because the default property can be represented by the object name **Err**, earlier code written using the **Err** function or **Err** statement doesn't have to be modified.

Use the following rules to determine whether to use the **Err.Description** function or the **ConnectionErrorMessage** property to retrieve the error text:

- If **Err.Number** returns the constant rcErrConnectionError, a general connection failure occurred. Use the **ConnectionErrorMessage** property to retrieve a text string that describes the specific cause of the error.
- If **Err.Number** does not return rcErrConnectionError, use **Err.Description** to retrieve a text string that describes the specific cause of the error. In this case, you can also use the constant returned by **Err.Number** to handle the error more specifically; for example, if you determine from the **Err.Number** value that a connection already exists (the constant rcErrAlreadyConnected), you could ask whether the user wants to disconnect and try again.

To see a list of the constants returned by **Err.Number** when a connection-related error occurs, open the Reflection Programming Help and search for *Connections, error handling* in the index. (See page 2 for instructions on how to view this Help.) Error constants are also listed in Rwinapi.txt and can also be viewed using the Visual Basic Editor's Object Browser.

Note: If **Err.Number** does not return rcErrConnectionError and you use the **ConnectionErrorMessage** property to retrieve the error text string, you will retrieve incorrect information. The **ConnectionErrorMessage** property always contains the text of the most recent general connection failure—that is, when **Err.Number** is rcErrConnectionError—or an empty string if no general connection failure occurred. Make sure to use **Err.Description** if **Err.Number** does not return rcErrConnectionError.

Example

In this example, a connection type of Telnet is configured, but a host name is not specified; this causes Reflection to prompt for the host name. After you enter a host name and click OK, the connection is attempted. If the connection fails for any reason, the macro's error handler is invoked. The error handler displays a message box with the error message text, and offers you the opportunity to try the connection again. Without the error handler, the macro simply stops if the connection cannot be established.

```
Sub ConnectionErrorDemo
   Dim theErrorMsg As String      ' To hold the error message.
   Dim theErrorText As String     ' To hold longer error text.
   Dim theResult As Integer       ' Value from Try Again dialog.

TryToConnect:
   On Error GoTo handler
   With Session
      If .Connected = True Then
         MsgBox "You are already connected."
         Exit Sub
      End If
      .ConnectionType = "TELNET"
      .ConnectionSettings = "Host ''"
      .Connect
   End With
   Exit Sub
```

```
Handler:
   With Session
      If Err.Number = rcErrConnectionError Then
         theErrorMsg = .ConnectionErrorMessage
      Else
         theErrorMsg = Err.Description
      End If
    End With

   theErrorText = "Connection error: " & theErrorMsg
   theErrorText = theErrorText & VbCr & VbLf & VbLf & "Try again?"
   theResult = MsgBox(theErrorText, vbOKCancel, "Connection Error")
   If theResult = VbCancel Then
      Exit Sub
   Else
      Resume TryToConnect
   End If
End Sub
```

CHAPTER 12

Reflection Basic Support

Reflection Basic is an Automation scripting language provided with earlier versions of Reflection. Reflection continues to support scripts developed using Reflection Basic. This chapter covers the following topics:

- Running Reflection Basic scripts
- Displaying the Script menu
- Comparing Reflection Basic to Visual Basic
- Why use Visual Basic?
- The Reflection object name (Application vs. Session)

Running Reflection Basic Scripts

Prior to version 7.0, Reflection menus included a Script menu for working with Reflection Basic scripts. Although the Script menu is no longer displayed by default, your existing scripts (*.rbs) can still run without any modification.

If you attached scripts to custom features (such as toolbar buttons or hotspots), you don't need to make any changes to your settings file; your scripts will continue to run exactly as they did before. Similarly, scripts that run automatically (such as connection scripts and scripts that are triggered by events) will continue to run as they did before.

Displaying the Script Menu

If you want to continue to maintain and edit Reflection Basic scripts, you can add the Script menu to your Reflection menu bar. The specific steps depend on the Reflection product you are using.

Using Reflection for IBM

This procedure restores the Script menu to the right of the new Macro menu on the Reflection menu bar:

1. Click **Menu** on the **Setup** menu to open the Menu Setup dialog box.
2. Click **Macro** under **Defined menu** to select this item.
3. In the **Available options** box, open **Additional items**, then select **Script**.
4. Click **Add After**. This adds the Script menu to the right of the Macro menu.
5. Click **OK** to close the dialog box.
6. Save your settings file to save this change.

Using Reflection for HP, UNIX and OpenVMS, or ReGIS Graphics

This procedure restores the Script menu to the right of the new Macro menu on the Reflection menu bar:

1. Click **Menu** on the **Setup** menu to open the Menu Setup dialog box.
2. Click **Macr&o** under **Defined menu** to select this item.
3. In the **Available options** box, open **Additional items**, open **Items from Version&6.x**, then select **Scri&pt**.
4. Click **Add After**. This adds the Script menu to the right of the Macro menu.
5. Click **OK** to close the dialog box.
6. On the **File** menu, click **Save** to save this change.

Comparing Reflection Basic to Visual Basic

Reflection Basic and Visual Basic for Applications are both dialects of the BASIC programming language. Reflection continues to support both dialects. If you run a Reflection Basic script, Reflection recognizes it as such and uses the Reflection Basic compiler to run the script; if you run a macro, Reflection uses Visual Basic to compile and run the macro. The following language elements are the same in Reflection Basic and VBA:

- Reflection methods and properties are identical. A small number of Reflection commands cause potential conflicts with equivalent Visual Basic commands. For more information about handling these situations, see *Keyword conflicts between Reflection and Visual Basic* in the Reflection Programming Help index. (See page 2 for instructions on how to view this Help.)
- Core BASIC language commands are generally the same. You can expect most statement and function syntax that works in Reflection Basic to work in Visual Basic. For more information about handling these situations, see *Reflection Basic, comparing Reflection Basic and Visual Basic* in the Reflection Programming Help index.

Differences between Reflection Basic and VBA include:

- In versions of Reflection prior to 7.0, all methods and properties acted on the **Application** object. If you are creating or maintaining Reflection Basic scripts, you should continue to use **Application** for the Reflection object name. If you are creating Visual Basic macros, use **Session** for the object name.
- If you want to create procedures that use Reflection events, you must use VBA. These events are not available from Reflection Basic.
- Dialog boxes in Visual Basic are created using forms. Dialog box statements used to create dialog boxes in Reflection Basic are not supported in VBA.
- Reflection Basic scripts are contained within separate files, and script execution always begins with the Main procedure. By default, macros are contained within a single project; each macro is a procedure within that project. Projects are saved when you save your settings files. (Macros can also be saved to separate files. See page 20 for more information.)

- In Reflection Basic, one script accesses information or procedures in another script using the **$Include** metacommand, the **Delcare** statement, or the **RunScript** method. In VBA, the **Public** and **Private** keywords are used to manage the availability (or scope) of elements within a project. With VBA, you can also add references to other projects to access code or forms in those projects.
- Visual Basic is a more object-oriented programming language than Reflection Basic. For example, using Visual Basic you can define your own object classes.

For more detailed information, open the Reflection Programming help and look up *Reflection Basic, comparing Reflection Basic and Visual Basic*. (See page 2 for instructions on how to view this help.)

Why Use Visual Basic?

If you have developed scripts using previous versions of Reflection Basic, you may be wondering if it's worthwhile to convert your scripts to Visual Basic macros. In making this decision, you may want to consider the following:

- Visual Basic can simplify management and distribution. Reflection Basic scripts must be distributed as a number of separate files, while macros can be distributed in a single settings file.
- Visual Basic for Applications is common to many programming applications, including Microsoft Office products. Your programming knowledge in any of these products will be immediately applicable to any other product that uses Visual Basic. In fact, you can write common code that you can share with other VBA applications.
- There are many resources available to help you answer questions and develop expertise with Visual Basic. These include books, magazines, training seminars, and a large number of programmers already familiar with this programming environment.

- The Visual Basic Editor is a much more powerful and flexible development environment than the Reflection Basic Editor. After you spend the initial time necessary to become familiar with its features, you'll find many tools that can help you develop your code more efficiently.
- The Visual Basic programming language is more powerful and more flexible than Reflection Basic. For example, programming language features not available in Reflection Basic include user forms and user-defined object classes.
- Visual Basic for Applications is a shared component. If you already have an application installed that uses VBA, Reflection uses that component.

The Reflection Object Name (Application vs. Session)

In versions of Reflection prior to 7.0, all methods and properties acted on the **Application** object. The newer **Session** object is now used in all help programming examples. However, if you are creating and/or maintaining Reflection Basic scripts, you should continue to use **Application** for the Reflection object name.

802.2 DLC 89

A

Adding references 79
Application object 27, 113
Auto Quick Info feature 36
Automation 71–79
  controlling other applications 72
  controlling Reflection 74
  examples 73
  OLEServerName property 77
  overview 71
Automation events 81

B

Bas files 19
Basic language programming 25

C

CheckBox control 46
Class Modules 27
Class modules 19
Closing custom dialog boxes 51
CLS files 19
Coax DFT 89
Code modules 19, 27
Code window 27, 35
ComboBox control 46
Command syntax 28
CommandButton control 46
Connect macros
  Reflection for HP 94
  Reflection for IBM 85
  Reflection for ReGIS Graphics 94
  Reflection for UNIX and OpenVMS 94
Connect method 85, 92
Connected property 85, 92
Connection Errors
  Reflection for HP 105
  Reflection for UNIX and OpenVMS 105
  Reflection forReGIS Graphics 105
Connection settings
  Reflection for IBM 87
Connections
  Reflection for HP 11, 91–107
  Reflection for IBM 9, 85–89
  Reflection for ReGIS Graphics 11, 91–107
  Reflection for UNIX and OpenVMS 11, 91–107
ConnectionSetting method 95
ConnectionSettings property 95, 96, 98, 101
ConnectionType property 96, 98
Constants 79
Context-sensitive help 24
Control structures 25
Controls in forms 46
CreateObject 72, 74, 76
Creating a custom dialog box 45
Creating a new form 45
Creating macros 37

D

Data types 25
Define event method 83
Demonstration hosts 9, 11
Dial method 104
Dialog boxes (also see Forms) 45–61
Disabled menu items 7
Disconnect method 85, 92

E

Early binding 74
Editing macros 39
Editing techniques 36
Err object 66
Error codes 70
Error function 67
Error handling 63–70
  connections 105
  examples 64, 68, 69
  inline 69
  resuming a procedure 68
  trapping an error 64
Events 25, 81–83
  Automation 81
  creating an event procedure 82
  methods and properties 83
Executable applications 24
Exercises
  ClipboardDemo (creating new macros) 40
  ColorDemo (forms) 52
  GetOpenFolder (forms) 57
  Login macro (using the recorder) 8

F

Forms 27, 45–61
  adding a new control 48
  adding controls 46
  closing 51
  creating 45
  exporting 19
  showing 50
  step-by-step exercises 52
  testing 50
Frame control 46
FRM files 19
Full module view 36
Functions 25

G

GetObject 72, 77

H

Help
  context-sensitive 24
  Reflection Programming 2, 28
  Visual Basic (Microsoft) 2
HP hosts 91

I

IBM hosts 85
Immediate window 36
Importing Visual Basic project files 19
Installation
  sample macros 3

K

Keyword/Value tokens 101

L

Label control 46
Language references 23
Late binding 74
ListBox control 46
Login exercise 8
Login macro
  Reflection for HP 11
  Reflection for IBM 9
  Reflection for ReGIS Graphics 11
  Reflection for UNIX and OpenVMS 11
Loops 25

M

Macros
  creating 37
  editing 39
  macro files 21
  naming rules 38
  recording 5, 8, 88, 102
  running 7
  samples 3
  saving 8
  sharing and managing 15–21
  stopping 7
Macros dialog box 7
Macros in macro files 20–21
  advantages 20
  compared to macros in settings files 21
  limitations 20
Managing macros 15–21
Microsoft Visual Basic Help 2
Modem Connections
  Reflection for HP 103
  Reflection for ReGIS Graphics 103
  Reflection for UNIX and OpenVMS 103
Modules 19, 27

N

Naming rules for macros 38
NewMacros module 27

O

Objects 25, 26, 27, 113
OLE Automation (see Automation)
OLEServerName property 77
On Error 64
OnEvent method 83
Opening custom dialog boxes 50
OpenVMS 91
Operators 25
Optional arguments 29
OptionButton control 46

P

Predefined constants 79
Procedures 25
Project components 26
Project Explorer 26, 33
Projects, overview 26
Properties 26
Properties window 34

Q

Quotation marks 29

R

Rbs files 109
Recording macros 5, 8, 88, 102
References
  adding references 18
  overview 16
  removing 19
  SharedMacros file 16
Reflection Basic 27, 67
  compared to Visual Basic 111
  displaying the Script menu 110
  running scripts 109
  support 109
Reflection language reference 24
Reflection object model 27
Reflection Programming Help 2, 28
Resume statement 68
RMA files 20
Rodecls.bas 79
Runtime errors 63
Rwinapi.txt 79

S

Sample macros 3
Saving macros 8
Scope 26
Script menu 110
SDLC 89
Session object 26, 29, 78, 113
Session setup
  Reflection for HP 95
  Reflection for IBM 87
  Reflection for ReGIS Graphics 95
  Reflection for UNIX and OpenVMS 95
SessionType property 89
Settings files 15, 21
SetupSession method 89
Shared component 24
SharedMacros file
  changing location 17
  overview 16
Sharing macros 15–21
SNA Engine 89
SNAEngineCFGFile property 89
Splitting the code window 36
Stopping macros 7
Sub procedures 25
Syntax 28, 36

T

Technical support
  phone number iii
TerminalModel property 89
TextBox control 46
ThisSession 26
Toolbox 47
TransportName property 89
TransportType property 89
Tutorial exercises
  ClipboardDemo (creating new macros) 40
  ColorDemo (forms) 52
  GetOpenFolder (forms) 57
  Login macro (using the recorder) 8

U

Unisys hosts 91
UNIX hosts 91
UseModemDialerV5 property 104
User forms, *see* Forms

V

VBA 23
View Settings dialog box 26
Visual Basic Editor 31–36
Visual Basic for Applications 23, 112
Visual Basic Help 2
Visual Basic language reference 23
Visual Basic Project files 19

W

Watch window 36
With statement 29
Wrqsna.rlf 89