No matter how carefully crafted your code, errors can (and probably will) occur. Ideally, Visual Basic procedures wouldn't need error-handling code at all. Unfortunately, sometimes files are mistakenly deleted, disk drives run out of space, or network drives disconnect unexpectedly. Such possibilities can cause run-time errors in your code. To handle these errors, you need to add error-handling code to your procedures.

Sometimes errors can also occur within your code; this type of error is commonly referred to as a *bug*. Minor bugs — for example, a cursor that doesn't behave as expected — can be frustrating or inconvenient. More severe bugs can cause an application to stop responding to commands, possibly requiring the user to restart the application, losing whatever work hasn't been saved.

The process of locating and fixing bugs in your application is known as *debugging*. Visual Basic provides several tools to help analyze how your application operates. These debugging tools are particularly useful in locating the source of bugs, but you can also use the tools to experiment with changes to your application or to learn how other applications work.

This chapter shows how to use the debugging tools included in Visual Basic and explains how to handle *run-time errors* — errors that occur while your code is running and that result from attempts to complete an invalid operation.

Note If you are using the Control Creation Edition of Visual Basic, some of the material covered in this document may not be applicable. With the full editions of Visual Basic you have the ability to create applications, ActiveX documents, and other types of components. Although some of the terminology may relate to application specific objects such as forms, in most cases the underlying concepts also apply to ActiveX control creation.

1

Contents

- Using Break Mode
- Running Selected Portions of Your Application
- Monitoring the Call Stack
- Testing Data and Procedures with the Immediate Window
- Special Debugging Considerations
- Tips for Debugging

2

Sample Application: Errors.vbp

Many of the code samples in this chapter are taken from the Errors.vbp sample application. If you installed the sample applications, you will find it in the \Errors subdirectory of the Visual Basic samples directory (\Vb\Pguide\Samples).

How to Handle Errors

Ideally, Visual Basic procedures wouldn't need error-handling code at all. Reality dictates that hardware problems or unanticipated actions by the user can cause run-time errors that halt your code, and there's usually nothing the user can do to resume running the application. Other errors might not interrupt code, but they can cause it to act unpredictably.

For example, the following procedure returns true if the specified file exists and false if it does not, but doesn't contain error-handling code:

```
Function FileExists (filename) As Boolean
    FileExists = (Dir(filename) <> "")
End Function
```

3

The Dir function returns the first file matching the specified file name (given with or without wildcard characters, drive name, or path); it returns a zero-length string if no matching file is found.

The code appears to cover either of the possible outcomes of the Dir call. However, if the drive letter specified in the argument is not a valid drive, the error "Device unavailable" occurs. If the specified drive is a floppy disk drive, this function will work correctly only if a disk is in the drive and the drive door is closed. If not, Visual Basic presents the error "Disk not ready" and halts execution of your code.

To avoid this situation, you can use the error-handling features in Visual Basic to intercept errors and take corrective action. (Intercepting an error is also known as *trapping* an error.) When an error occurs, Visual Basic sets the various properties of the error object, Err, such as an error number, a description, and so on. You can use the Err object and its properties in an error-handling routine so that your application can respond intelligently to an error situation.

For example, device problems, such as an invalid drive or an empty floppy disk drive, could be handled by the following code:

—2

```
Function FileExists (filename) As Boolean
    Dim Msg As String
    ' Turn on error trapping so error handler responds
    ' if any error is detected.
    On Error GoTo CheckError
        FileExists = (Dir(filename) <> "")
        ' Avoid executing error handler if no error
        ' occurs.
        Exit Function

CheckError:             ' Branch here if error occurs.
    ' Define constants to represent intrinsic Visual
    ' Basic error codes.
    Const mnErrDiskNotReady = 71, _
    mnErrDeviceUnavailable = 68
    ' vbExclamation, vbOK, vbCancel, vbCritical, and
    ' vbOKCancel are constants defined in the VBA type
    ' library.
    If (Err.Number = MnErrDiskNotReady) Then
        Msg = "Put a floppy disk in the drive "
        Msg = Msg & "and close the door."
        ' Display message box with an exclamation mark
        ' icon and with OK and Cancel buttons.
        If MsgBox(Msg, vbExclamation & vbOKCancel) = _
        vbOK Then
            Resume
        Else
            Resume Next
        End If
    ElseIf Err.Number = MnErrDeviceUnavailable Then
        Msg = "This drive or path does not exist: "
        Msg = Msg & filename
        MsgBox Msg, vbExclamation
        Resume Next
    Else
        Msg = "Unexpected error #" & Str(Err.Number)
        Msg = Msg & " occurred: " & Err.Description
        ' Display message box with Stop sign icon and
        ' OK button.
        MsgBox Msg, vbCritical
        Stop
    End If
    Resume
End Function
```

4

In this code, the Err object's Number property contains the number associated with the run-time error that occurred; the Description property contains a short description of the error.

When Visual Basic generates the error "Disk not ready," this code presents a message telling the user to choose one of two buttons — OK or Cancel. If the user chooses OK, the Resume statement returns control to the statement at which the error occurred

—3

and attempts to re-execute that statement. This succeeds if the user has corrected the problem; otherwise, the program returns to the error handler.

If the user chooses Cancel, the Resume Next statement returns control to the statement following the one at which the error occurred (in this case, the Exit Function statement).

Should the error "Device unavailable" occur, this code presents a message describing the problem. The Resume Next statement then causes the function to continue execution at the statement following the one at which the error occurred.

If an unanticipated error occurs, a short description of the error is displayed and the code halts at the Stop statement.

The application you create can correct an error or prompt the user to change the conditions that caused the error. To do this, use techniques such as those shown in the preceding example. The next section discusses these techniques in detail.

For More Information See "Guidelines for Complex Error Handling" in "The Error-Handling Hierarchy" later in this chapter for an explanation of how to use the Stop statement.

5

Designing an Error Handler

An *error handler* is a routine for trapping and responding to errors in your application. You'll want to add error handlers to any procedure where you anticipate the possibility of an error (you should assume that any Basic statement can produce an error unless you explicitly know otherwise). The process of designing an error handler involves three steps:

1. Set, or *enable*, an error trap by telling the application where to branch to (which error-handling routine to execute) when an error occurs.

   1The On Error statement enables the trap and directs the application to the label marking the beginning of the error-handling routine.

   2In the Errors.vpb sample application, the FileExists function contains an error-handling routine named CheckError.

2. Write an error-handling routine that responds to all errors you can anticipate. If control actually branches into the trap at some point, the trap is then said to be *active*.

   3The CheckError routine handles the error using an If...Then...Else statement that responds to the value in the Err object's Number property, which is a numeric code corresponding to a Visual Basic error. In the example, if "Disk not ready" is generated, a message prompts the user to close the drive door. A different message is displayed if the "Device unavailable" error occurs. If any other error is generated, the appropriate description is displayed and the program stops.

—4

3. Exit the error-handling routine.

4In the case of the "Disk not ready" error, the Resume statement makes the code branch back to the statement where the error occurred. Visual Basic then tries to re-execute that statement. If the situation has not changed, then another error occurs and execution branches back to the error-handling routine.

5In the case of the "Device unavailable" error, the Resume Next statement makes the code branch to the statement following the one at which the error occurred.

1

Details on how to perform these steps are provided in the remainder of this topic. Refer to the FileExists function example as you read through these steps.

Setting the Error Trap

An error trap is enabled when Visual Basic executes the On Error statement, which specifies an error handler. The error trap remains enabled while the procedure containing it is active —that is, until an Exit Sub, Exit Function, Exit Property, End Sub, End Function, or End Property statement is executed for that procedure. While only one error trap can be enabled at any one time in any given procedure, you can create several alternative error traps and enable different ones at different times. You can also disable an error trap by using a special case of the On Error statement — On Error GoTo 0.

To set an error trap that jumps to an error-handling routine, use a On Error GoTo *line* statement, where *line* indicates the label identifying the error-handling code. In the FileExists function example, the label is CheckError. (Although the colon is part of the label, it isn't used in the On Error GoTo *line* statement.)

For More Information For more information about disabling error handling, see the topic, "Turning Off Error Handling," later in this chapter.

6

Writing an Error-Handling Routine

The first step in writing an error-handling routine is adding a line label to mark the beginning of the error handling routine. The line label should have a descriptive name and must be followed by a colon. A common convention is to place the error-handling code at the end of the procedure with an Exit Sub, Exit Function, or Exit Property statement immediately before the line label. This allows the procedure to avoid executing the error-handling code if no error occurs.

The body of the error handling routine contains the code that actually handles the error, usually in the form of a Case or If…Then…Else statement. You need to determine which errors are likely to occur and provide a course of action for each, for example, prompting the user to insert a disk in the case of a "Disk not ready" error. An option should always be provided to handle any unanticipated errors by using the Else or Case Else clause — in the case of the FileExists function example, this option warns the user then ends the application.

The Number property of the Err object contains a numeric code representing the most recent run-time error. By using the Err object in combination with the Select Case or If...Then...Else statement, you can take specific action for any error that occurs.

Note The string contained in the Err object's Description property explains the error associated with the current error number. The exact wording of the description may vary among different versions of Microsoft Visual Basic. Therefore, use Err.Number, rather than Err.Description, to identify the specific error that occurred.

7

Exiting an Error-Handling Routine

The FileExists function example uses the Resume statement within the error handler to re-execute the statement that originally caused the error, and uses the Resume Next statement to return execution to the statement following the one at which the error occurred. There are other ways to exit an error-handling routine. Depending on the circumstances, you can do this using any of the statements shown in the following table.

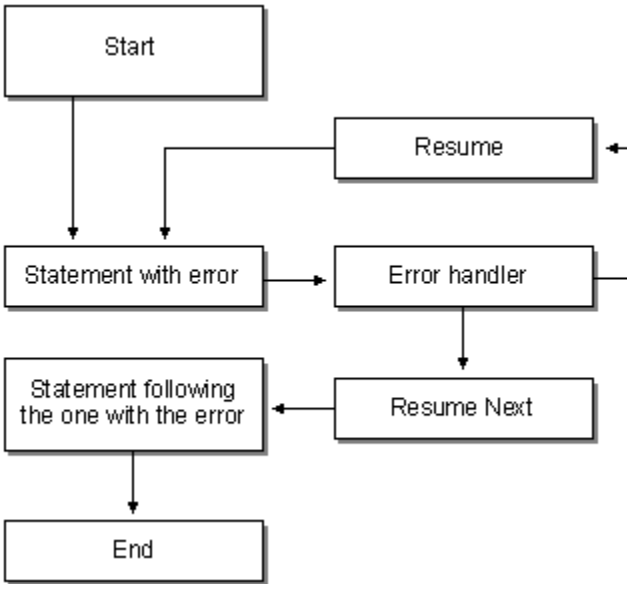
Statement	Description
Resume [0]	Program execution resumes with the statement that caused the error or the most recently executed call out of the procedure containing the error-handling routine. Use it to repeat an operation after correcting the condition that caused the error.
Resume Next	Resumes program execution at the statement immediately following the one that caused the error. If the error occurred outside the procedure that contains the error handler, execution resumes at the statement immediately following the call to the procedure wherein the error occurred, if the called procedure does not have an enabled error handler.
Resume *line*	Resumes program execution at the label specified by *line*, where *line* is a line label (or nonzero line number) that must be in the same procedure as the error handler.
Err.Raise Number:= *number*	Triggers a run-time error. When this statement is executed within the error-handling routine, Visual Basic searches the calls list for another error-handling routine. (The *calls list* is the chain of procedures invoked to arrive at the current point of execution. See the section, "The Error-Handling Hierarchy," later in this chapter.)

8

The Difference Between Resume and Resume Next Statements

The difference between Resume and Resume Next is shown in Figure 13.1.

Figure 13.1 Program flow with Resume and Resume Next



2

Generally, you would use Resume whenever the error handler can correct the error, and Resume Next when the error handler cannot. You can write an error handler so that the existence of a run-time error is never revealed to the user or to display error messages and allow the user to enter corrections.

For example, the Function procedure in the following code example uses error handling to perform "safe" division on its arguments without revealing errors that might occur. The errors that can occur when performing division are:

Error	Cause
"Division by zero"	Numerator is nonzero, but the denominator is zero.
"Overflow"	Both numerator and denominator are zero (during floating-point division).
"Illegal procedure call"	Either the numerator or the denominator is a nonnumeric value (or can't be considered a numeric value).

9

In all three cases, the following Function procedure traps these errors and returns Null:

```
Function Divide (numer, denom) as Variant
    Dim Msg as String
    Const mnErrDivByZero = 11, mnErrOverFlow = 6
    Const mnErrBadCall = 5
    On Error GoTo MathHandler
        Divide   = numer / denom
        Exit Function
MathHandler:
```

—7

```
    If Err.Number = MnErrDivByZero Or _
    Err.Number = ErrOverFlow _
    Or Err = ErrBadCall Then
        Divide = Null      ' If error was Division by
                           ' zero, Overflow, or Illegal
                           ' procedure call, return Null.
    Else
        ' Display unanticipated error message.
        Msg = "Unanticipated error " & Err.Number
        Msg = Msg & ": " & Err.Description
        MsgBox Msg, vbExclamation
    End If              ' In all cases, Resume Next
                        ' continues execution at
    Resume Next         ' the Exit Function statement.
End Function
```

10

Resuming Execution at a Specified Line

Resume Next can also be used where an error occurs within a loop, and you need to restart the operation. Or, you can use Resume *line*, which returns control to a specified line label.

The following example illustrates the use of the Resume *line* statement. A variation on the FileExists example shown earlier, this function allows the user to enter a file specification that the function returns if the file exists.

```
Function VerifyFile As String
    Const mnErrBadFileName = 52, _
    mnErrDriveDoorOpen = 71
    Const mnErrDeviceUnavailable = 68, _
    mnErrInvalidFileName = 64
    Dim strPrompt As String, strMsg As String, _
    strFileSpec As String
    strPrompt = "Enter file specification to check:"
StartHere:
    strFileSpec = "*.*"    ' Start with a default
                           ' specification.
    strMsg = strMsg & vbCRLF & strPrompt
    ' Let the user modify the default.
    strFileSpec = InputBox(strMsg, "File Search", _
    strFileSpec, 100, 100)
    ' Exit if user deletes default.
    If strFileSpec = "" Then Exit Function
    On Error GoTo Handler
        VerifyFile = Dir(strFileSpec)
        Exit Function
Handler:
    Select Case Err.Number ' Analyze error code and
                           ' load message.
        Case ErrInvalidFileName, ErrBadFileName
            strMsg = "Your file specification was "
            strMsg = strMsg & "invalid; try another."
        Case MnErrDriveDoorOpen
            strMsg = "Close the disk drive door and "
```

—8

```
            strMsg = strMsg & "try again."
        Case MnErrDeviceUnavailable
            strMsg = "The drive you specified was not "
            strMsg = strMsg & "found. Try again."
        Case Else
            Dim intErrNum As Integer
            intErrNum = Err.Number
            Err.Clear                   ' Clear the Err object.
            Err.Raise Number:= intErrNum          ' Regenerate
                                                  ' the error.
    End Select
    Resume StartHere   ' This jumps back to StartHere
                       ' label so the user can try
                       ' another file name.
End Function
```

11

If a file matching the specification is found, the function returns the file name. If no matching file is found, the function returns a zero-length string. If one of the anticipated errors occurs, a message is assigned to the strMsg variable and execution jumps back to the label StartHere. This gives the user another chance to enter a valid path and file specification.

If the error is unanticipated, the Case Else segment regenerates the error so that the next error handler in the calls list can trap the error. This is necessary because if the error wasn't regenerated, the code would continue to execute at the Resume StartHere line. By regenerating the error you are in effect causing the error to occur again; the new error will be trapped at the next level in the call stack.

For More Information For more details, see the topic, "The Error Handling Hierarchy" later in this chapter.

Note Although using Resume *line* is a legitimate way to write code, a proliferation of jumps to line labels can render code difficult to understand and debug.

12

The Error Handling Hierarchy

An *enabled* error handler is one that was activated by executing an On Error statement and hasn't yet been turned off — either by an On Error GoTo 0 statement or by exiting the procedure where it was enabled. An *active* error handler is one in which execution is currently taking place. To be active, an error handler must first be enabled, but not all enabled error handlers are active. For example, after a Resume statement, a handler is deactivated but still enabled.

When an error occurs within a procedure lacking an enabled error-handling routine, or within an active error-handling routine, Visual Basic searches the calls list for another enabled error-handling routine. The calls list is the sequence of calls that leads to the currently executing procedure; it is displayed in the Call Stack dialog box. You can display the Call Stack dialog box only when in break mode (when you pause

the execution of your application), by selecting the View, Call Stack menu item or by pressing CTRL+L.

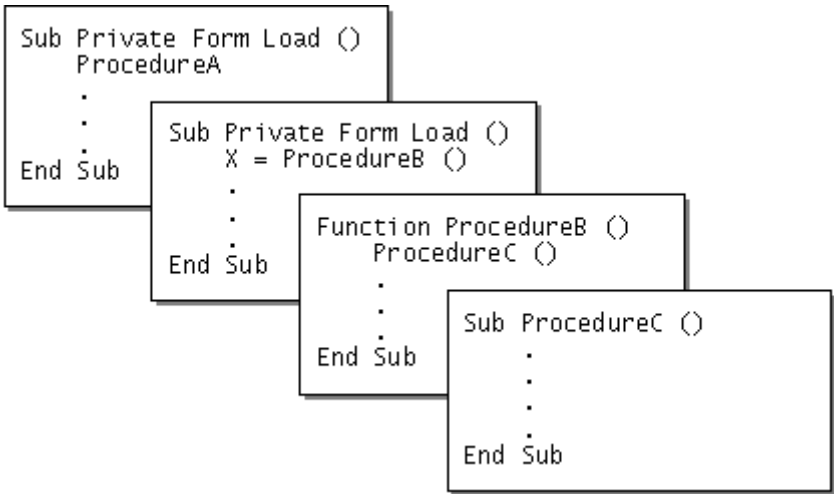
Searching the Calls List

Suppose the following sequence of calls occurs, as shown in Figure 13.2:

4. An event procedure calls Procedure A.
5. Procedure A calls Procedure B.
6. Procedure B calls Procedure C.

3

Figure 13.2 A sequence of calls



4

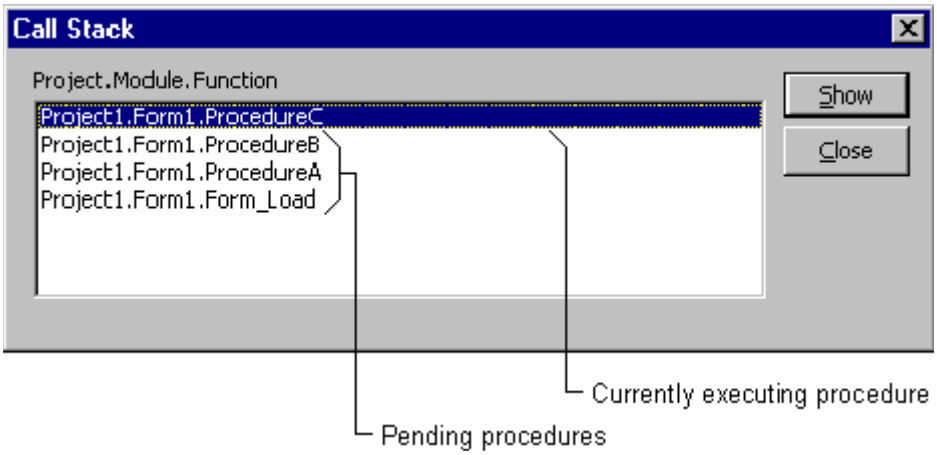
While Procedure C is executing, the other procedures are pending, as shown in the calls list in the Call Stack dialog box.

For More Information For more information, see "Monitoring the Call Stack" later in this chapter.

13

Figure 13.3 shows the calls list displayed in the Call Stack dialog box.

Figure 13.3 The calls list when procedures are pending



5

If an error occurs in Procedure C and this procedure doesn't have an enabled error handler, Visual Basic searches backward through the pending procedures in the calls list — first Procedure B, then Procedure A, then the initial event procedure (but no farther) — and executes the first enabled error handler it finds. If it doesn't encounter an enabled error handler anywhere in the calls list, it presents a default unexpected error message and halts execution.

If Visual Basic finds an enabled error-handling routine, execution continues in that routine as if the error had occurred in the same procedure that contains the error handler. If a Resume or a Resume Next statement is executed in the error-handling routine, execution continues as shown in the following table.

Statement	Result
Resume	The call to the procedure that Visual Basic just searched is re-executed. In the calls list given earlier, if Procedure A has an enabled error handler that includes a Resume statement, Visual Basic re-executes the call to Procedure B.
Resume Next	Execution returns to the statement following the last statement executed in that procedure. This is the statement following the call to the procedure that Visual Basic just searched back through. In the calls list given earlier, if Procedure A has an enabled error handler that includes a Resume Next statement, execution returns to the statement after the call to Procedure B.

14

Notice that the statement executed is in the procedure *where the error-handling procedure is found*, not necessarily in the procedure where the error occurred. If you don't take this into account, your code may perform in ways you don't intend. To make the code easier to debug, you can simply go into break mode whenever an error occurs, as explained in the section, "Turning Off Error Handling," later in this chapter.

If the error handler's range of errors doesn't include the error that actually occurred, an unanticipated error can occur within the procedure with the enabled error handler. In such a case, the procedure could execute endlessly, especially if the error handler executes a Resume statement. To prevent such situations, use the Err object's Raise method in a Case Else statement in the handler. This actually generates an error within the error handler, forcing Visual Basic to search through the calls list for a handler that can deal with the error.

In the VerifyFile procedure example in the Errors.vbp sample application, the number originally contained in Err.Number is assigned to a variable, intErrNum, which is then passed as an argument to the Err object's Raise method in a Case Else statement, thereby generating an error. When such an error occurs within an active error handler, the search back through the calls list begins.

Allocating Errors to Different Handlers

The effect of the search back through the calls list is hard to predict, because it depends on whether Resume or Resume Next is executed in the handler that processes the error successfully. Resume returns control to the most recently executed call out of the procedure containing the error handler. Resume Next returns control to whatever statement immediately follows the most recently executed call out of the procedure containing the error handler.

For example, in the calls list shown in Figure 13.3, if Procedure A has an enabled error handler and Procedure B and C don't, an error occurring in Procedure C will be handled by Procedure A's error handler. If that error handler uses a Resume statement, upon exit, the program continues with a call to Procedure B. However, if Procedure A's error handler uses a Resume Next statement, upon exit, the program will continue with whatever statement in Procedure A follows the call to Procedure B. In both cases the error handler does not return directly to either the procedure or the statement where the error originally occurred.

Guidelines for Complex Error Handling

When you write large Visual Basic applications that use multiple modules, the error-handling code can get quite complex. Keep these guidelines in mind:

- While you are debugging your code, use the Err object's Raise method to regenerate the error in all error handlers for cases where no code in the handler deals with the specific error. This allows your application to try to correct the error in other error-handling routines along the calls list. It also ensures that Visual Basic will display an error message if an error occurs that your code doesn't handle. When you test your code, this technique helps you uncover the errors you aren't handling adequately. However, in a stand-alone .exe file, you should be cautious: If you execute the Raise method and no other procedure traps the error, your application will terminate execution immediately, without any QueryUnload or Unload events occurring.

- Use the Clear method if you need to explicitly clear the Err object after handling an error. This is necessary when using inline error handling with On Error Resume Next. Visual Basic calls the Clear method automatically whenever it executes any type of Resume statement, Exit Sub, Exit Function, Exit Property, or any On Error statement.
- If you don't want another procedure in the calls list to trap the error, use the Stop statement to force your code to terminate. Using Stop lets you examine the context of the error while refining your code in the development environment.

  Caution Be sure to remove any Stop statements before you create an .exe file. If a stand-alone Visual Basic application (.exe) encounters a Stop statement, it treats it as an End statement and terminates execution immediately, without any QueryUnload or Unload events occurring.
- Write a fail-safe error-handling procedure that all your error handlers can call as a last resort for errors they cannot handle. This fail-safe procedure can perform an orderly termination of your application by unloading forms and saving data.

For More Information See the "Inline Error Handling," "Design Time, Run Time, and Break Mode," and "Testing Error Handling by Generating Errors" topics later in this chapter.

Testing Error Handling by Generating Errors

Simulating errors is useful when you are testing your applications, or when you want to treat a particular condition as being equivalent to a Visual Basic run-time error. For example, you might be writing a module that uses an object defined in an external application, and want errors returned from the object to be handled as actual Visual Basic errors by the rest of your application.

In order to test for all possible errors, you may need to generate some of the errors in your code. You can generate an error in your code with the Raise method:

object.**Raise** *argumentlist*

The *object* argument is usually Err, Visual Basic's globally defined error object. The *argumentlist* argument is a list of named arguments that can be passed with the method. The VerifyFile procedure in the Errors.vbp sample application uses the following code to regenerate the current error in an error handler:

```
Err.Raise Number:=intErrNum
```

In this case, intErrNum is a variable that contains the error number which triggered the error handler. When the code reaches a Resume statement, the Clear method of the

Err object is invoked. It is necessary to regenerate the error in order to pass it back to the previous procedure on the call stack.

You can also simulate any Visual Basic run-time error by supplying the error code for that error:

```
Err.Raise Number:=71    ' Simulate "Disk Not Ready"
                        ' error.
```

18

Defining Your Own Errors

Sometimes you may want to define errors in addition to those defined by Visual Basic. For example, an application that relies on a modem connection might generate an error when the carrier signal is dropped. If you want to generate and trap your own errors, you can add your error numbers to the vbObjectError constant.

The vbObjectError constant reserves the numbers ranging from its own offset to its offset + 512. Using a number higher than this will ensure that your error numbers will not conflict with future versions of Visual Basic or other Microsoft Basic products. ActiveX controls may also define their own error numbers. To avoid conflicts with them, consult the documentation for controls you use in your application.

To define your own error numbers, you add constants to the Declarations section of your module:

```
' Error constants
Const gLostCarrier = 1 + vbObjectError + 512
Const gNoDialTone = 2 + vbObjectError + 512
```

19

You can then use the Raise method as you would with any of the intrinsic errors. In this case, the description property of the Err object will return a standard description — "Application-defined or object defined error." To provide your own error description, you will need to add it as a parameter to the Raise method.

Inline Error Handling

You may be accustomed to programming in a language that doesn't raise exceptions — in other words, it doesn't interrupt your code's execution by generating exceptions when errors occur, but instead records errors for you to check later. The C programming language works in this manner, and you may sometimes find it convenient to follow this practice in your Visual Basic code.

When you check for errors immediately after each line that may cause an error, you are performing *inline error handling.* This topic explains the different approaches to inline error handling, including:

- Writing functions and statements that return error numbers when an error occurs.
- Raising a Visual Basic error in a procedure and handling the error in an inline error handler in the calling procedure.

- Writing a function to return a Variant data type, and using the Variant to indicate to the calling procedure that an error occurred.

7

Returning Error Numbers

There are a number of ways to return error numbers. The simplest way is to create functions and statements that return an error number, instead of a value, if an error occurs. The following example shows how you can use this approach in the FileExists function example, which indicates whether or not a particular file exists.

```
Function FileExists (p As String) As Long
    If Dir (p) <> " " Then
        FileExists = conSuccess ' Return a constant
                                                    ' indicating the
    Else                                            ' file exists.
        FileExists = conFailure   ' Return failure
                                                    ' constant.
    End If
End Function

Dim ResultValue As Long
ResultValue = FileExists ("C:\Testfile.txt")
If ResultValue = conFailure Then
    .
    .    ' Handle the error.
    .
Else
    .
    .    ' Proceed with the program.
    .
End If
```

20

The key to inline error handling is to test for an error immediately after each statement or function call. In this manner, you can design a handler that anticipates exactly the sort of error that might arise and resolve it accordingly. This approach does not require that an actual run-time error arise. This becomes useful when working with API and other DLL procedures which do not raise Visual Basic exceptions. Instead, these procedures indicate an error condition, either in the return value, or in one of the arguments passed to the procedures; check the documentation for the procedure you are using to determine how these procedures indicate an error condition.

Handling Errors in the Calling Procedure

Another way to indicate an error condition is to raise a Visual Basic error in the procedure itself, and handle the error in an inline error handler in the calling procedure. The next example shows the same FileExists procedure, raising an error number if it is not successful. Before calling this function, the On Error Resume Next statement sets the values of the Err object properties when an error occurs, but without trying to execute an error-handling routine.

The On Error Resume Next statement is followed by error-handling code. This code can check the properties of the Err object to see if an error occurred. If Err.Number doesn't contain zero, an error has occurred, and the error-handling code can take the appropriate action based on the values of the Err object's properties.

```
Function FileExists (p As String)
    If Dir (p) <> " " Then
        Err.Raise conSuccess    ' Return a constant
                                ' indicating the
    Else                        'file exists.
        Err.Raise conFailure' Raise error number
                                ' conFailure.
    End If
End Function

Dim ResultValue As Long
On Error Resume Next
ResultValue = FileExists ("C:\Testfile.txt")
If Err.Number = conFailure Then
    .
    .    ' Handle the error.
    .
Else
    .
    .    ' Continue program.
    .
End If
```

21

The next example uses both the return value and one of the passed arguments to indicate whether or not an error condition resulted from the function call.

```
Function Power (X As Long, P As Integer, _
ByRef Result As Integer)As Long
    On Error GoTo ErrorHandler
    Result = x^P
    Exit Function
ErrorHandler:
    Power = conFailure
End Function

' Calls the Power function.
Dim lngReturnValue As Long, lngErrorMaybe As Long
lngErrorMaybe = Power (10, 2, lngReturnValue)
If lngErrorMaybe Then
    .
    .    ' Handle the error.
    .
Else
    .
    .    ' Continue program.
    .
End If
```

22

If the function was written simply to return either the result value or an error code, the resulting value might be in the range of error codes, and your calling procedure would not be able to distinguish them. By using both the return value and one of the passed arguments, your program can determine that the function call failed, and take appropriate action.

Using Variant Data Types

Another way to return inline error information is to take advantage of the Visual Basic Variant data type and some related functions. A Variant has a tag that indicates what type of data is contained in the variable, and it can be tagged as a Visual Basic error code. You can write a function to return a Variant, and use this tag to indicate to the calling procedure that an error has occurred.

The following example shows how the Power function can be written to return a Variant.

```
Function Power (X As Long, P As Integer) As Variant
    On Error GoTo ErrorHandler
    Power = x^P
    Exit Function

ErrorHandler:
    Power = CVErr(Err.Number) ' Convert error code to
                                          ' tagged Variant.
End Function

' Calls the Power function.
Dim varReturnValue As Variant
varReturnValue = Power (10, 2)
If IsError (varReturnValue) Then
    .
    .    ' Handle the error.
    .
Else
    .
    .    ' Continue program.
    .
End If
```

23

Centralized Error Handling

When you add error-handling code to your applications, you'll quickly discover that you're handling the same errors over and over. With careful planning, you can reduce code size by writing a few procedures that your error-handling code can call to handle common error situations.

The following FileErrors function procedure shows a message appropriate to the error that occurred and, where possible, allows the user to choose a button to specify what action the program should take next. It then returns code to the procedure that called

it. The value of the code indicates which action the program should take. Note that user-defined constants such as MnErrDeviceUnavailable must be defined somewhere (either globally, or at the module level of the module containing the procedure, or within the procedure itself). The constant vbExclamation is defined in the Visual Basic (VB) object library, and therefore does not need to be declared.

```
Function FileErrors As Integer
    Dim intMsgType As Integer, strMsg As String
    Dim intResponse As Integer
    ' Return Value         Meaning
    ' 0                        Resume
    ' 1                        Resume Next
    ' 2                        Unrecoverable error
    ' 3                        Unrecognized error
    intMsgType = vbExclamation
    Select Case Err.Number
        Case MnErrDeviceUnavailable   ' Error 68.
            strMsg = "That device appears unavailable."
            intMsgType = vbExclamation + 4
        Case MnErrDiskNotReady        ' Error 71.
            strMsg = "Insert a disk in the drive "
            strMsg = strMsg & "and close the door."
            intMsgType = vbExclamation + 4
        Case MnErrDeviceIO            ' Error 57.
            strMsg = "Internal disk error."
            intMsgType = vbExclamation + 4
        Case MnErrDiskFull            ' Error 61.
            strMsg = "Disk is full. Continue?"
            intMsgType = vbExclamation + 3
        ' Error 64 & 52.
        Case ErrBadFileName, ErrBadFileNameOrNumber
            strMsg = "That filename is illegal."
            intMsgType = vbExclamation + 4
        Case ErrPathDoesNotExi        ' Error 76.
            strMsg = "That path doesn't exist."
            intMsgType = vbExclamation + 4
        Case ErrBadFileMode           ' Error 54.
            strMsg = "Can't open your file for that "
            strMsg = strMsg & "type of access."
            intMsgType = vbExclamation + 4
        Case ErrFileAlreadyOpen     ' Error 55.
            strMsg = "This file is already open."
            intMsgType = vbExclamation + 4
        Case ErrInputPastEndOfFile ' Error 62.
            strMsg = "This file has a nonstandard "
            strMsg = strMsg & "end-of-file marker, "
            strMsg = strMsg & "or an attempt was made “
            strMsg = strMsg & “to read beyond "
            strMsg = strMsg & "the end-of-file marker."
            intMsgType = vbExclamation + 4
        Case Else
            FileErrors = 3
            Exit Function
```

```
    End Select
    intResponse = MsgBox (strMsg, intMsgType, _
    "Disk Error")
    Select Case intRresponse
        Case 1, 4          ' OK, Retry buttons.
            FileErrors = 0
        Case 5             ' Ignore button.
            FileErrors = 1
        Case 2, 3          ' Cancel, End buttons.
            FileErrors = 2
        Case Else
            FileErrors = 3
    End Select
End Function
```

24

This procedure handles common file and disk-related errors. If the error is not related to disk Input/Output, it returns the value 3. The procedure that calls this procedure should then either handle the error itself, regenerate the error with the Raise method, or call another procedure to handle it.

Note As you write larger applications, you'll find that you are using the same constants in several procedures in various forms and modules. Making those constants public and declaring them in a single standard module may better organize your code and save you from typing the same declarations repeatedly.

25

You can simplify error handling by calling the FileErrors procedure wherever you have a procedure that reads or writes to disk. For example, you've probably used applications that warn you if you attempt to replace an existing disk file. Conversely, when you try to open a file that doesn't exist, many applications warn you that the file does not exist and ask if you want to create it. In both instances, errors can occur when the application passes the file name to the operating system.

The following checking routine uses the value returned by the FileErrors procedure to decide what action to take in the event of a disk-related error.

```
Function ConfirmFile (FName As String, _
Operation As Integer) As Integer
' Parameters:
' Fname: File to be checked for and confirmed.
' Operation: Code for sequential file access mode
' (Output, Input, and so on).
' Note that the procedure works for binary and random
' access because messages are conditioned on Operation
' being <> to certain sequential modes.
' Return values:
' 1       Confirms operation will not cause a problem.
' 0       User decided not to go through with operation.
    Const conSaveFile = 1, conLoadFile = 2
    Const conReplaceFile = 1, conReadFile = 2
    Const conAddToFile = 3, conRandomFile = 4
    Const conBinaryFile = 5
```

```
Dim intConfirmation As Integer
Dim intAction As Integer
Dim intErrNum As Integer, varMsg As Variant

On Error GoTo ConfirmFileError ' Turn on the error
                                                    ' trap.
FName = Dir(FName)             ' See if the file exists.
On Error GoTo 0                ' Turn error trap off.
' If user is saving text to a file that already
' exists...
If FName <> "" And Operation = conReplaceFile Then
    varMsg = "The file " & FName &
    varMsg = varMsg & "already exists on " & vbCRLF
    varMsg = varMsg & "disk. Saving the text box "
    varMsg = varMsg & & vbCRLF
    varMsg = varMsg & "contents to that file will "
    varMsg = varMsg & "destroy the file's current "
    varMsg = varMsg & "contents, " & vbCRLF _
    varMsg = varMsg & "replacing them with the "
    varMsg = varMsg & "text from the text box."
    varMsg = varMsg & vbCRLF & vbCRLF
    varMsg = varMsg & "Choose OK to replace file, "
    varMsg = varMsg & "Cancel to stop."
    intConfirmation = MsgBox(varMsg, 65, _
    "File Message")
' If user wants to load text from a file that
' doesn't exist.
ElseIf FName = "" And Operation = conReadFile Then
    varMsg = "The file " & FName
    varMsg = varMsg & " doesn't exist." & vbCRLF
    varMsg = varMsg & _
    "Would you like to create and varMsg = varMsg & "then edit it?" _
    & vbCRLF
    varMsg = varMsg & vbCRLF & "Choose OK to "
    varMsg = varMsg & "create file, Cancel to stop."
    intConfirmation = MsgBox(varMsg, 65, _
    "File Message")
' If FName doesn't exist, force procedure to return
' 0 by setting
' intConfirmation = 2.
ElseIf FName = "" Then
    If Operation = conRandomFile Or _
    Operation = conBinaryFile Then
        intConfirmation = 2
    End If
' If the file exists and operation isn't
' successful,
' intConfirmation = 0 and procedure returns 1.
End If
' If no box was displayed, intConfirmation = 0;
' if user chose OK, in either case,
' intConfirmation = 1 and ConfirmFile should
' return 1 to confirm that the intended operation
' is OK. If intConfirmation > 1, ConfirmFile should
```

```
    ' return 0, because user doesn't want to go through
    ' with the operation...
    If intConfirmation > 1 Then
        ConfirmFile = 0
    Else
        ConfirmFile = 1
        If Confirmation = 1 Then
            ' User wants to create file.
            If Operation = conLoadFile Then
                ' Assign conReplaceFile so caller will
                ' understand action that will be taken.
                Operation = conReplaceFile
            End If
        ' Return code confirming action to either
        ' replace existing file or create new one.
        End If
    End If
Exit Function
ConfirmFileError:
intAction = FileErrors
    Select Case intAction
        Case 0
            Resume
        Case 1
            Resume Next
        Case 2
            Exit Function
        Case Else
            intErrNum = Err.Number
            Err.Raise Number:=intErrNum
            Err.Clear
    End Select
End Function
```

26

The ConfirmFile procedure receives a specification for the file whose existence will be confirmed, plus information about which access mode will be used when an attempt is made to actually open the file. If a sequential file is to be saved (conReplaceFile), and a file is found that already has that name (and will therefore be overwritten), the user is prompted to confirm that overwriting the file is acceptable.

If a sequential file is to be opened (conReadFile) and the file is not found, the user is prompted to confirm that a new file should be created. If the file is being opened for random or binary access, its existence or nonexistence is either confirmed (return value 1) or refuted (return value 0). If an error occurs in the call to Dir, the FileErrors procedure is called to analyze the error and prompt the user for a reasonable course of action.

Turning Off Error Handling

If an error trap has been enabled in a procedure, it is automatically disabled when the procedure finishes executing. However, you may want to turn off an error trap in a

procedure while the code in that procedure is still executing. To turn off an enabled error trap, use the On Error GoTo 0 statement. Once Visual Basic executes this statement, errors are detected but not trapped within the procedure. You can use On Error GoTo 0 to turn off error handling anywhere in a procedure — even within an error-handling routine itself.

For example, try single stepping, using Step Into, through a procedure such as this:

```
Sub ErrDemoSub ()
    On Error GoTo SubHandler  ' Error trapping is
                                          ' enabled.
        ' Errors need to be caught and corrected here.
        ' The Kill function is used to delete a file.
        Kill "Oldfile.xyz"
    On Error GoTo 0      ' Error trapping is turned off
                              ' here.
        Kill "Oldfile.xyz"
    On Error GoTo SubHandler  ' Error trapping is
                                          ' enabled again.
        Kill "Oldfile.xyz"
    Exit Sub
SubHandler:               ' Error-handling routine goes here.
    MsgBox "Caught error."
    Resume Next
End Sub
```

27

For More Information To learn how to use the Step Into feature, see "Running Selected Portions of Your Application" later in this chapter.

28

Debugging Code with Error Handlers

When you are debugging code, you may find it confusing to analyze its behavior when it generates errors that are trapped by an error handler. You could comment out the On Error line in each module in the project, but this is also cumbersome.

Instead, while debugging, you could turn off error handlers so that every time there's an error, you enter break mode.

To disable error handlers while debugging

1 From the **Tools** menu, choose **Options** and click the **General** tab.

2 Select the **Break on All Errors** option, and then choose **OK**.

8

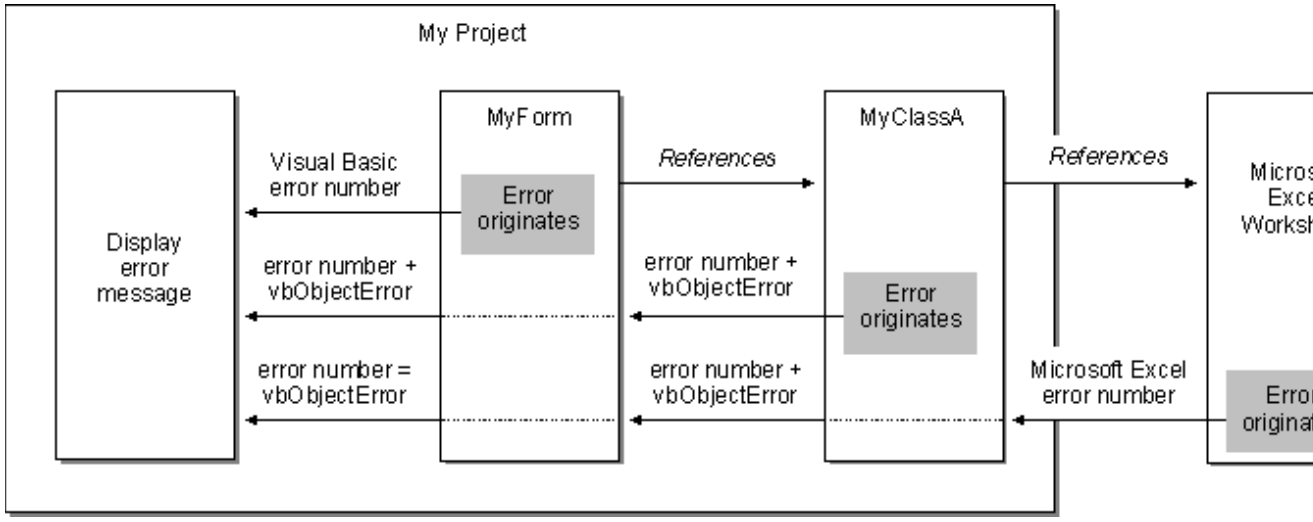
With this option selected, when an error occurs anywhere in the project, you will enter break mode and the Code window will display the code where the error occurred.

If this option is not selected, an error may or may not cause an error message to be displayed, depending on where the error occurred. For example, it may have been raised by an external object referenced by your application. If it does display a message, it may be meaningless, depending on where the error originated.

Error Handling with ActiveX Components

In applications that use one or more objects, it becomes more difficult to determine where an error occurs, particularly if it occurs in another application's object. For example, Figure 13.4 shows an application that consists of a form module, that references a class module, that in turn references a Microsoft Excel Worksheet object.

Figure 13.4 Regenerating errors between forms, classes, and ActiveX components



9

If the Worksheet object does not handle a particular error arising in the Worksheet, but regenerates it instead, Visual Basic will pass the error to the referencing object, MyClassA. Visual Basic automatically remaps untrapped errors arising in objects outside of Visual Basic as error code 440.

The MyClassA object can either handle the error (which is preferable), or regenerate it. The interface specifies that any object regenerating an error that arises in a referenced object should not simply propagate the error (pass as error code 440), but should instead remap the error number to something meaningful. When you remap the error, the number can either be a number defined by Visual Basic that indicates the error condition, if your handler can determine that the error is similar to a defined Visual Basic error (for instance, overflow or division by zero), or an undefined error number. Add the new number to the intrinsic Visual Basic constant vbObjectError to notify other handlers that this error was raised by your object.

Whenever possible, a class module should try to handle every error that arises within the module itself, and should also try to handle errors that arise in an object it references that are not handled by that object. However, there are some errors that it cannot handle because it cannot anticipate them. There are also cases where it is more

appropriate for the referencing object to handle the error, rather than the referenced object.

When an error occurs in the form module, Visual Basic raises one of the predefined Visual Basic error numbers.

Note If you are creating a public class, be sure to clearly document the meaning of each non-Visual Basic error-handler you define. (Public classes cannot be created in the Standard Edition.) Other programmers who reference your public classes will need to know how to handle errors raised by your objects.

29

When you regenerate an error, leave the Err object's other properties unchanged. If the raised error is not trapped, the Source and Description properties can be displayed to help the user take corrective action.

Handling Errors in Objects

A class module could include the following error handler to accommodate any error it might trap, regenerating those it is unable to resolve:

```
MyServerHandler:
    Select Case ErrNum
        Case 7        ' Handle out-of-memory error.
            .
            .
            .
        Case 440          ' Handle external object error.
            Err.Raise Number:=vbObjectError + 9999
        ' Error from another Visual Basic object.
        Case Is > vbObjectError and Is < vbObjectError _
        + 65536
            ObjectError = ErrNum
        Select Case ObjectError
            ' This object handles the error, based on
            ' error code documentation for the object.
            Case vbObjectError + 10
            .
            .
            .
            Case Else
                ' Remap error as generic object error and
                ' regenerate.
                Err.Raise Number:=vbObjectError + 9999
            End Select
        Case Else
            ' Remap error as generic object error and
            ' regenerate.
            Err.Raise Number:=vbObjectError + 9999
    End Select
    Err.Clear
    Resume Next
```

30

The Case 440 statement traps errors that arise in a referenced object outside the Visual Basic application. In this example, the error is simply propagated using the value 9999, because it is difficult for this type of centralized handler to determine the cause of the error. When this error is raised, it is generally the result of a fatal automation error (one that would cause the component to end execution), or because an object didn't correctly handle a trapped error. Error 440 shouldn't be propagated unless it is a fatal error. If this trap were written for an inline handler as discussed previously in the topic, "Inline Error Handling," it might be possible to determine the cause of the error and correct it.

The statement

```
Case Is > vbObjectError and Is < vbObjectError + 65536
```


traps errors that originate in an object within the Visual Basic application, or within the same object that contains this handler. Only errors defined by objects will be in the range of the vbObjectError offset.

The error code documentation provided for the object should define the possible error codes and their meaning, so that this portion of the handler can be written to intelligently resolve anticipated errors. The actual error codes may be documented without the vbObjectError offset, or they may be documented after being added to the offset, in which case the Case Else statement should subtract vbObjectError, rather than add it. On the other hand, object errors may be constants, shown in the type library for the object, as shown in the Object Browser. In that case, use the error constant in the Case Else statement, instead of the error code.

Any error not handled should be regenerated with a new number, as shown in the Case Else statement. Within your application, you can design a handler to anticipate this new number you've defined. If this were a public class (not available in the Standard Edition), you would also want to include an explanation of the new error-handling code in your application's documentation.

The last Case Else statement traps and regenerates any other errors that are not trapped elsewhere in the handler. Because this part of the trap will catch errors that may or may not have the vbObjectError constant added, you should simply remap these errors to a generic "unresolved error" code. That code should be added to vbObjectError, indicating to any handler that this error originated in the referenced object.

Debugging Error Handlers in ActiveX Components

When you are debugging an application that has a reference to an object created in Visual Basic or a class defined in a class module, you may find it confusing to determine which object generates an error. To make this easier, you can select the Break in Class Module option on the General tab of the Options dialog box (available from the Tools menu). With this option selected, an error in a class module or an

object in another application or project that is running in Visual Basic will cause that class to enter the debugger's break mode, allowing you to analyze the error. An error arising in a compiled object will not display the Immediate window in break mode; rather, such errors will be handled by the object's error handler, or trapped by the referencing module.

Approaches to Debugging

The debugging techniques presented in this chapter use the analysis tools provided by Visual Basic. Visual Basic cannot diagnose or fix errors for you, but it does provide tools to help you analyze how execution flows from one part of the procedure to another, and how variables and property settings change as statements are executed. Debugging tools let you look inside your application to help you determine what happens and why.

Visual Basic debugging support includes breakpoints, break expressions, watch expressions, stepping through code one statement or one procedure at a time, and displaying the values of variables and properties. Visual Basic also includes special debugging features, such as edit-and-continue capability, setting the next statement to execute, and procedure testing while the application is in break mode.

For More Information For a quick overview of Visual Basic debugging, see "Tips for Debugging" later in this chapter.

32

Kinds of Errors

To understand how debugging is useful, consider the three kinds of errors you can encounter:

- Compile errors
- Run-time errors
- Logic errors

10

Compile Errors

Compile errors result from incorrectly constructed code. If you incorrectly type a keyword, omit some necessary punctuation, or use a Next statement without a corresponding For statement at design time, Visual Basic detects these errors when you compile the application.

Compile errors include errors in syntax. For example, you could have a statement as follows:

```
Left
```

33

Left is a valid word in the Visual Basic language, but without an object, it doesn't meet the syntax requirements for that word (*object*.Left). If you have selected the

Auto Syntax Check option in the Editor tab on the Options dialog box, Visual Basic will display an error message as soon as you enter a syntax error in the Code window.

To set the Auto Syntax Check option

3 From the **Tools** menu, select **Options**, and click the **Editor** tab on the **Options** dialog box.

4 Select **Auto Syntax Check**.

11

For More Information See the section "Avoiding Bugs" later in this chapter for other techniques to use to avoid errors in your code.

34

Run-Time Errors

Run-time errors occur while the application is running (and are detected by Visual Basic) when a statement attempts an operation that is impossible to carry out. An example of this is division by zero. Suppose you have this statement:

```
Speed = Miles / Hours
```

35

If the variable Hours contains zero, the division is an invalid operation, even though the statement itself is syntactically correct. The application must run before it can detect this error.

For More Information You can include code in your application to trap and handle run-time errors when they occur. For information on dealing with run-time errors, see "How to Handle Errors" earlier in this chapter.

36

Logic Errors

Logic errors occur when an application doesn't perform the way it was intended. An application can have syntactically valid code, run without performing any invalid operations, and yet produce incorrect results. Only by testing the application and analyzing results can you verify that the application is performing correctly.

How Debugging Tools Help

Debugging tools are designed to help you with:

- Logic and run-time errors.
- Observing the behavior of code that has no errors.

12

For instance, an incorrect result may be produced at the end of a long series of calculations. In debugging, the task is to determine what and where something went wrong. Perhaps you forgot to initialize a variable, chose the wrong operator, or used an incorrect formula.

There are no magic tricks to debugging, and there is no fixed sequence of steps that works every time. Basically, debugging helps you understand what's going on while

your application runs. Debugging tools give you a snapshot of the current state of your application, including:

- Appearance of the user interface (UI).
- Values of variables, expressions, and properties.
- Active procedure calls.

13

The better you understand how your application is working, the faster you can find bugs.

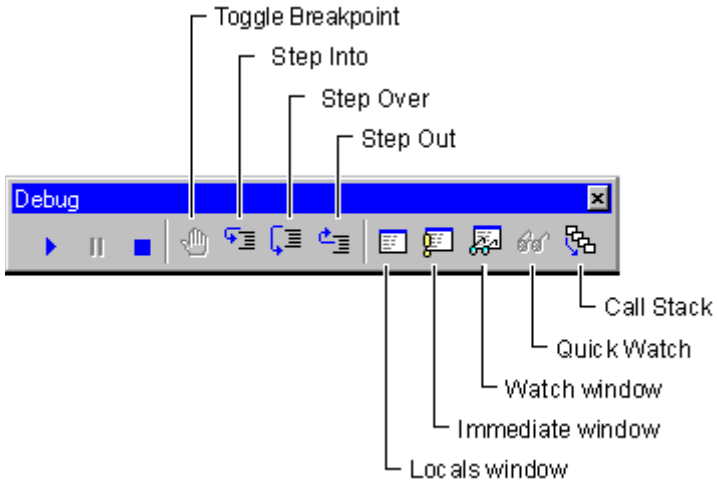
For More Information For more details on viewing and testing variables, expressions, properties, and active procedure calls, see "Testing Data and Procedures with the Immediate Window" and "Monitoring the Call Stack" later in this chapter.

37

The Debug Toolbar

Among its many debugging tools, Visual Basic provides several buttons on the optional Debug toolbar that are very helpful. Figure 13.5 shows these tools. To display the Debug toolbar, right-click on the Visual Basic toolbar and select the Debug option.

Figure 13.5 The Debug toolbar



14

The following table briefly describes each tool's purpose. The topics in this chapter discuss situations where each of these tools can help you debug or analyze an application more efficiently.

Debugging tool	Purpose
Breakpoint	Defines a line in the Code window where Visual Basic suspends execution of the application.

Step Into	Executes the next executable line of code in the application and steps into procedures.
Step Over	Executes the next executable line of code in the application without stepping into procedures.
Step Out	Executes the remainder of the current procedure and breaks at the next line in the calling procedure.
Locals Window	Displays the current value of local variables.
Immediate Window	Allows you to execute code or query values while the application is in break mode.
Watch window	Displays the values of selected expressions.
Quick Watch	Lists the current value of an expression while the application is in break mode.
Call Stack	While in break mode, presents a dialog box that shows all procedures that have been called but not yet run to completion.

38

For More Information The debugging tools are only necessary if there are bugs in your application. See "Avoiding Bugs" later in this chapter.

Avoiding Bugs

There are several ways to avoid creating bugs in your applications:

- Design your applications carefully by writing down the relevant events and the way your code will respond to each one. Give each event procedure and each general procedure a specific, well-defined purpose.
- Include numerous comments. As you go back and analyze your code, you'll understand it much better if you state the purpose of each procedure in comments.
- Explicitly reference objects whenever possible. Declare objects as they are listed in the Classes/Modules box in the Object Browser, rather than using a Variant or the generic Object data types.
- Develop a consistent naming scheme for the variables and objects in your application.
- One of the most common sources of errors is incorrectly typing a variable name or confusing one control with another. You can use Option Explicit to avoid misspelling variable names. For more information on requiring explicit variable declaration, see "Introducing Variables, Constants, and Data Types" in "Programming Fundamentals."

15

Design Time, Run Time, and Break Mode

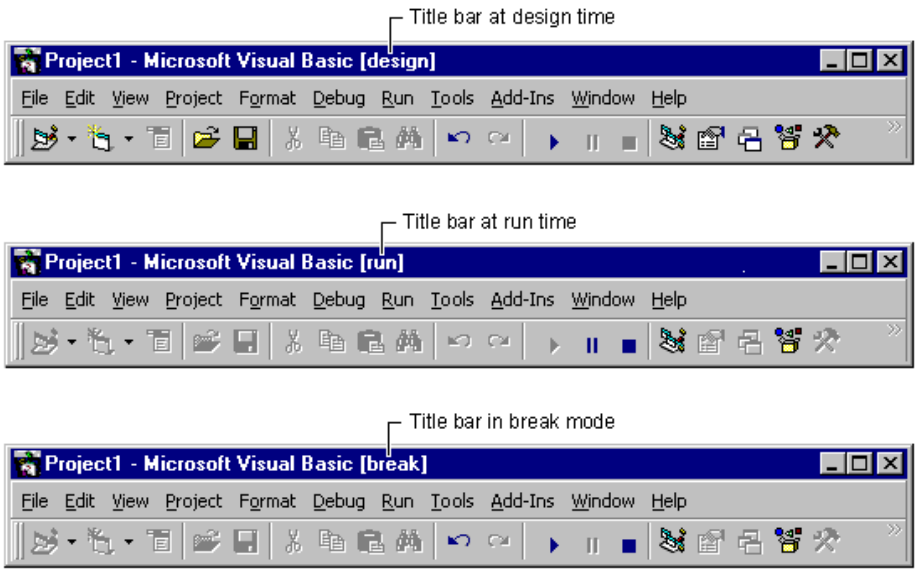
To test and debug an application, you need to understand which of three modes you are in at any given time. You use Visual Basic at design time to create an application,

and at run time to run it. This chapter introduces *break mode*, which suspends the execution of the program so you can examine and alter data.

Identifying the Current Mode

The Visual Basic title bar always shows you the current mode. Figure 13.6 shows the title bar for design time, run time, and break mode.

Figure 13.6 Identifying the current mode with the Visual Basic title bar



16

The characteristics of the three modes are listed in the following table.

Mode	Description
Design time	Most of the work of creating an application is done at design time. You can design forms, draw controls, write code, and use the Properties window to set or view property settings. You cannot execute code or use debugging tools, except for setting breakpoints and creating watch expressions. From the Run menu, choose Start, or click the Run button to switch to run time. If your application contains code that executes when the application starts, choose Step Into from the Run menu (or press F8) to place the application in break mode at the first executable statement.
Run time	When an application takes control, you interact with the application the same way a user would. You can view code, but you cannot change it. From the Run menu, choose End, or click the End button to switch back to design time.
Break mode	From the Run menu, choose Break, click the Break button, or press

CTRL+BREAK to switch to break mode.

Execution is suspended while running the application. You can view and edit code (choose Code from the View menu, or press F7), examine or modify data, restart the application, end execution, or continue execution from the same point.

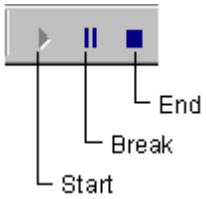
You can set breakpoints and watch expressions at design time, but other debugging tools work only in break mode. See "Using Break Mode" later in this chapter.

40 39

Using the Toolbar to Change Modes

The toolbar provides three buttons that let you change quickly from one mode to another. These buttons appear in Figure 13.7.

Figure 13.7 Start, Break, and End buttons on the toolbar



17

Whether any of these buttons is available depends on whether Visual Basic is in run-time mode, design-time mode, or break mode. The following table lists the buttons available for different modes.

Mode	Toolbar buttons available
Design time	Start
Run time	Break, End
Break	Continue, End (in break mode, the Start button becomes the Continue button)

41

Using the Debugging Windows

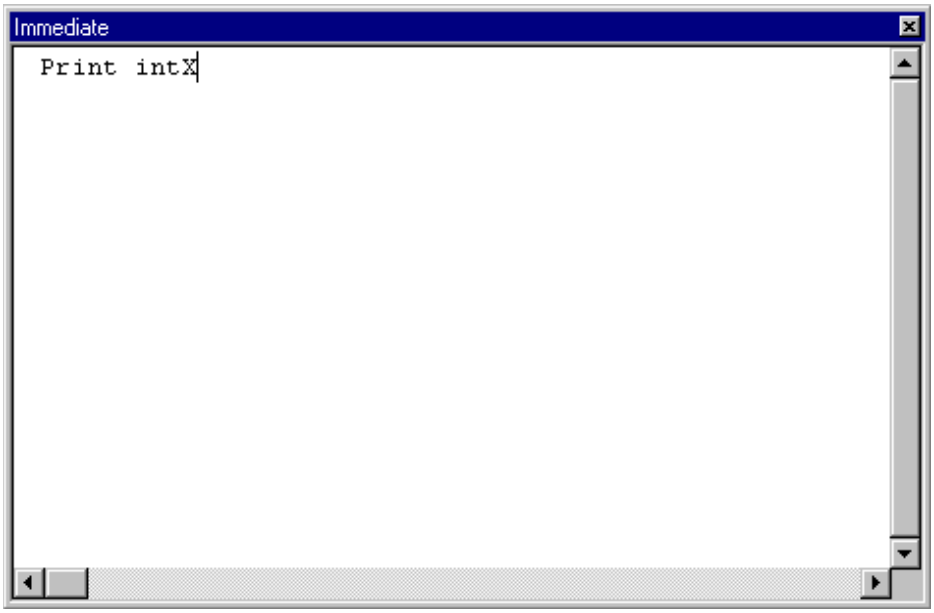
Sometimes you can find the cause of a problem by executing portions of code. More often, however, you'll also have to analyze what's happening to the data. You might isolate a problem in a variable or property with an incorrect value, and then have to determine how and why that variable or property was assigned an incorrect value.

With the debugging windows, you can monitor the values of expressions and variables while stepping through the statements in your application. There are three debugging windows: the Immediate window, the Watch window, and the Locals window

- The *Immediate window* shows information that results from debugging statements in your code, or that you request by typing commands directly into the window.

1Figure 13.8 The Immediate window

1



18

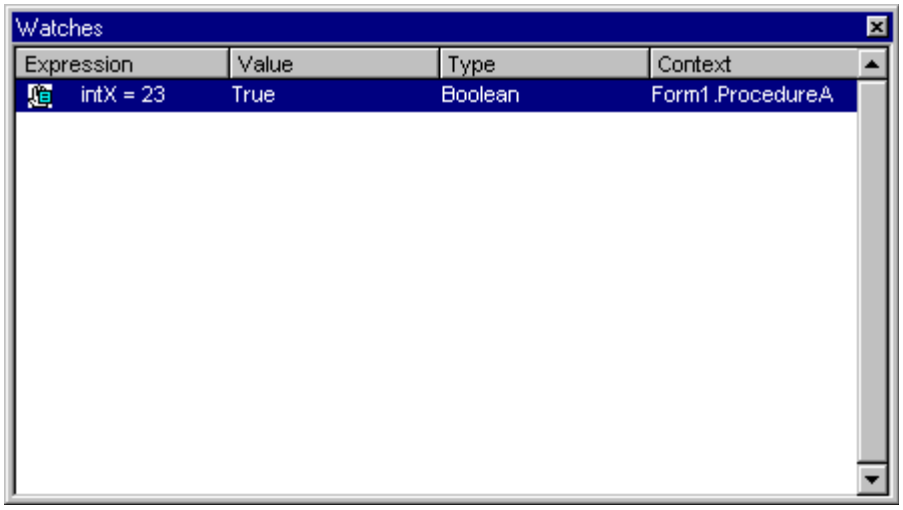
6**For More Information** To learn more about the Immediate window, see "Testing Data and Procedures with the Immediate Window" later in this chapter.

42

- The *Watch window* shows the current *watch expressions*, which are expressions whose values you decide to monitor as the code runs. A *break expression* is a watch expression that will cause Visual Basic to enter break mode when a certain condition you define becomes true. In the Watch window, the Context column indicates the procedure, module, or modules in which each watch expression is evaluated. The Watch window can display a value for a watch expression only if the current statement is in the specified context. Otherwise, the Value column shows a message indicating the statement is not in context. . To access the Watch window, select Watch Window from the View menu. Figure 13.9 shows the Watch window.

2Figure 13.9 The Watch window

2



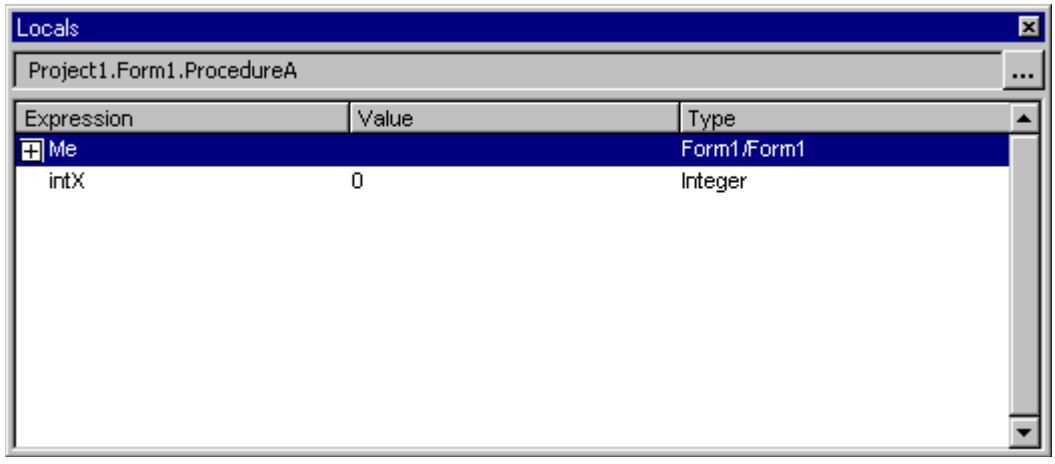
19

7**For More Information** To learn more about the Watch window, see "Monitoring Data with Watch Expressions" later in this chapter.

43

- The *Locals window* shows the value of any variables within the scope of the current procedure. As the execution switches from procedure to procedure, the contents of the Locals window changes to reflect only the variables applicable to the current procedure. To access the Locals window, select Locals Window from the View menu. Figure 13.10 shows the Locals window.

3Figure 13.10 The Locals window



20

—33

The current procedure and form (or module) determine which variables can be displayed according to the scoping rules presented in "Understanding the Scope of Variables" in "Programming Fundamentals." For example, suppose the Immediate window indicates that Form1 is the current form. In this case, you can display any of the form-level variables in Form1. You can also use Debug.Print to examine local variables of the procedure displayed in the Code window. (You can always examine the value of a public variable.) For more information about printing information in the Immediate window, see "Testing data and Procedures with the Immediate Window" later in this chapter.

Using Break Mode

At design time, you can change the design or code of an application, but you cannot see how your changes affect the way the application runs. At run time, you can watch how the application behaves, but you cannot directly change the code.

Break mode halts the operation of an application and gives you a snapshot of its condition at any moment. Variable and property settings are preserved, so you can analyze the current state of the application and enter changes that affect how the application runs. When an application is in break mode, you can:

- Modify code in the application.
- Observe the condition of the application's interface.
- Determine which active procedures have been called.
- Watch the values of variables, properties, and statements.
- Change the values of variables and properties.
- View or control which statement the application will run next.
- Run Visual Basic statements immediately.
- Manually control the operation of the application.

21

Note You can set breakpoints and watch expressions at design time, but other debugging tools work only in break mode.

44

Entering Break Mode at a Problem Statement

When debugging, you may want the application to halt at the place in the code where you think the problem might have started. This is one reason Visual Basic provides breakpoints and Stop statements. A *breakpoint* defines a statement or set of conditions at which Visual Basic automatically stops execution and puts the application in break mode without running the statement containing the breakpoint. See "Using Stop Statements" later in this chapter for a comparison of Stop statements and breakpoints.

You can enter break mode manually if you do any of the following while the application is running:

- Press CTRL+BREAK.
- Choose Break from the Run menu.
- Click the Break button on the toolbar.

22

It's possible to break execution when the application is idle (when it is between processing of events). When this happens, execution does not stop at a specific line, but Visual Basic switches to break mode anyway.

You can also enter break mode automatically when any of the following occurs:

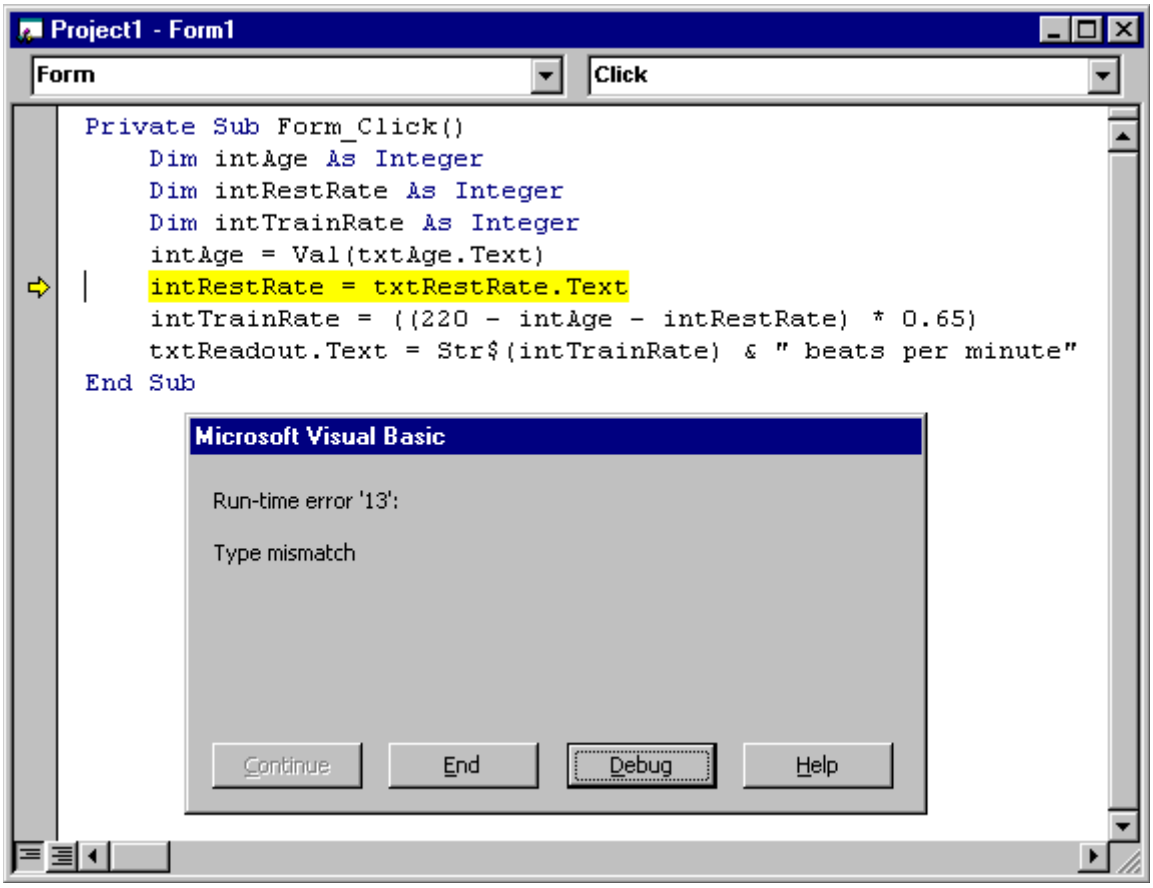
- A statement generates an untrapped run-time error.
- A statement generates a run-time error and Break on All Errors was selected in the General tab on the Options dialog box (available from the Tools menu).
- A break expression defined in the Add Watch dialog box changes or becomes true, depending on how you defined it.
- Execution reaches a line with a breakpoint.
- Execution reaches a Stop statement.

23

Fixing a Run-Time Error and Continuing

Some run-time errors result from simple oversights when entering code; these errors are easily fixed. Frequent errors include misspelled names and mismatched properties or methods with objects — for example, trying to use the Clear method on a text box, or the Text property with a file list box. Figure 13.11 shows a run-time error message.

Figure 13.11 Run-time errors halt execution



24

Often you can enter a correction and continue program execution with the same line that halted the application, even though you've changed some of the code. Simply choose Continue from the Run menu or click the Continue button on the toolbar. As you continue running the application, you can verify that the problem is fixed.

If you select the Break on All Errors option on the General tab on the Options dialog box (available from the Tools menu), Visual Basic disables error handlers in code, so that when a statement generates a run-time error, Visual Basic enters break mode. If Break on All Errors is not selected, and if an error handler exists, it will intercept code and take corrective action.

Some changes (most commonly, changing variable declarations or adding new variables or procedures) require you to restart the application. When this happens, Visual Basic presents a message that asks if you want to restart the application.

Monitoring Data with Watch Expressions

As you debug your application, a calculation may not produce the result you want or problems might occur when a certain variable or property assumes a particular value or range of values. Many debugging problems aren't immediately traceable to a single statement, so you may need to observe the behavior of a variable or expression throughout a procedure.

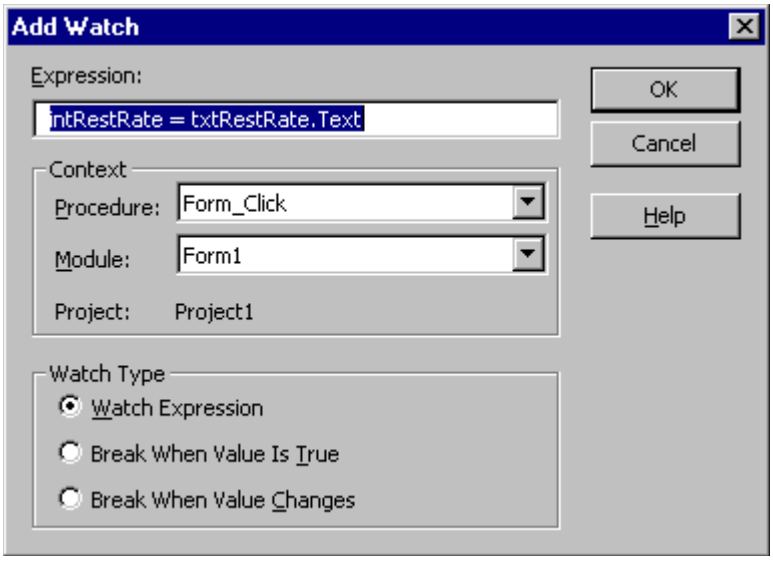
Visual Basic automatically monitors watch expressions — expressions that you define — for you. When the application enters break mode, these watch expressions appear in the Watch window, where you can observe their values.

You can also direct watch expressions to put the application into break mode whenever the expression's value changes or equals a specified value. For example, instead of stepping through perhaps tens or hundreds of loops one statement at a time, you can use a watch expression to put the application in break mode when a loop counter reaches a specific value. Or you may want the application to enter break mode each time a flag in a procedure changes value.

Adding a Watch Expression

You can add a watch expression at design time or in break mode. You use the Add Watch dialog box (shown in Figure 13.12) to add watch expressions.

Figure 13.12 The Add Watch dialog box



25

The following table describes the Add Watch dialog box.

Component	Description
Expression box	Where you enter the expression that the watch expression evaluates. The expression is a variable, a property, a function call, or any other valid expression.
Context option group	Sets the scope of variables watched in the expression. Use if you have variables of the same name with different scope. You can also restrict the scope of variables in watch expressions to a specific procedure or to a specific form or module, or you can have it apply to the entire application by selecting All Procedures and All Modules. Visual Basic can evaluate a variable in a narrow context more quickly.
Watch Type option group	Sets how Visual Basic responds to the watch expression. Visual Basic can watch the expression and display its value in the Watch window when the application enters break mode. Or you can have the application enter break mode automatically when the expression evaluates to a true (nonzero) statement or each time the value of the expression changes.

45

To add a watch expression

5 From the **Debug** menu, choose **Add Watch**.

6 The current expression (if any) in the Code Editor will appear in the **Expression** box on the **Add Watch** dialog box. If this isn't the expression you want to watch, enter the expression to evaluate in the **Expression** box.

7 If necessary, set the scope of the variables to watch.

8If you select the **Procedure** or **Module** option under **Context**, select a procedure, form, or module name from the appropriate list box.

8 If necessary, select an option button in the **Watch Type** group to determine how you want Visual Basic to respond to the watch expression.

9 Choose **OK**.

26

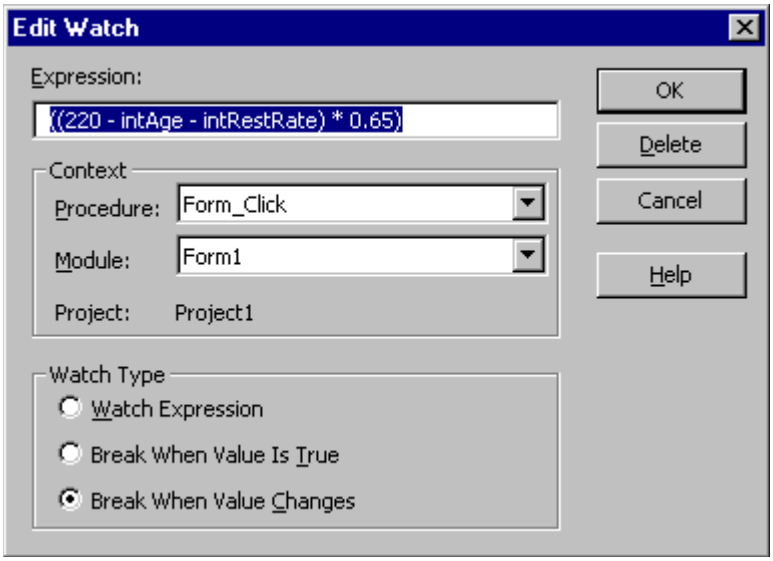
Note You can also add an expression by dragging and dropping from the Code Editor to the Watch window.

46

Editing or Deleting a Watch Expression

The Edit Watch dialog box, shown in Figure 13.13, lists all the current watch expressions. You can edit and delete any watch listed in the Watch window.

Figure 13.13 The Edit Watch dialog box



27

To edit a watch expression

10 In the **Watch** window, double click the watch expression you want to edit.

9– or –

10Select the watch expression you want to edit and choose **Edit Watch** from the **Debug** menu.

11 The **Edit Watch** dialog box is displayed and is identical to the Add Watch dialog box except for the title bar and the addition of a Delete button.

12 Make any changes to the expression, the scope for evaluating variables, or the watch type.

13 Choose **OK**.

28

To delete a watch expression

14 In the **Watch** window, select the watch expression you want to delete.

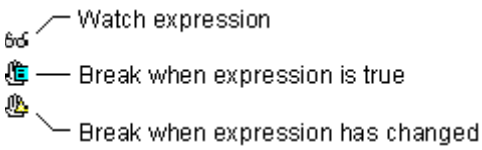
15 Press the DELETE key.

29

Identifying Watch Types

At the left edge of each watch expression in the Watch window is an icon identifying the watch type of that expression. Figure 13.14 defines the icon for each of the three watch types.

Figure 13.14 Watch type icons

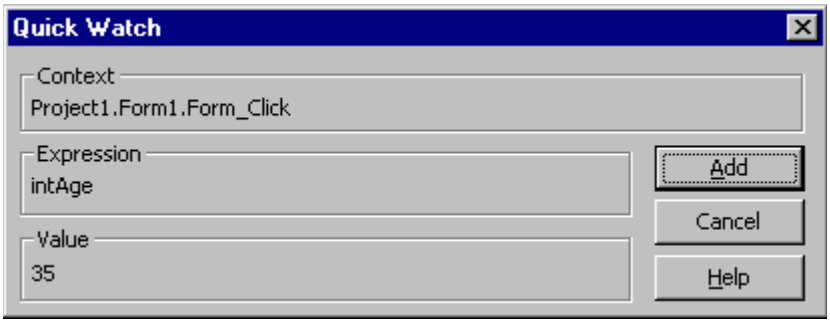


30

Using Quick Watch

While in break mode, you can check the value of a property, variable, or expression for which you have not defined a watch expression. To check such expressions, use the Quick Watch dialog box, shown in Figure 13.15.

Figure 13.15 The Quick Watch dialog box



31

The Quick Watch dialog box shows the value of the expression you select from the Code window. To continue watching this expression, click the Add button; the Watch window, with relevant information from the Quick Watch dialog box already entered, is displayed. If Visual Basic cannot evaluate the value of the current expression, the Add button is disabled.

To display the Quick Watch dialog box

16 Select a watch expression in the Code window.

17 Click the **Quick Watch** button on the **Debug** toolbar. (To display the Debug toolbar, right-click on the Visual Basic toolbar and select the **Debug** option.)

11– or –

12Press SHIFT+F9.

13– or –

14From the **Debug** menu, choose **Quick Watch**.

18 If you want to add a watch expression based on the expression in the **Quick Watch** dialog box, choose the **Add** button.

32

Using a Breakpoint to Selectively Halt Execution

At run time, a breakpoint tells Visual Basic to halt just before executing a specific line of code. When Visual Basic is executing a procedure and it encounters a line of code with a breakpoint, it switches to break mode.

You can set or remove a breakpoint in break mode or at design time, or at run time when the application is idle.

To set or remove a breakpoint

19 In the Code window, move the insertion point to the line of code where you want to set or remove a breakpoint.

15– or –

16Click in the margin on the left edge of the Code window next to the line where you want to set or remove a breakpoint.

20 From the **Debug** menu, choose **Toggle Breakpoint**.

17– or –

18Click the **Toggle Breakpoint** button on the **Debug** toolbar. (To display the Debug toolbar, right-click on the Visual Basic toolbar and select the **Debug** option.)

19– or –

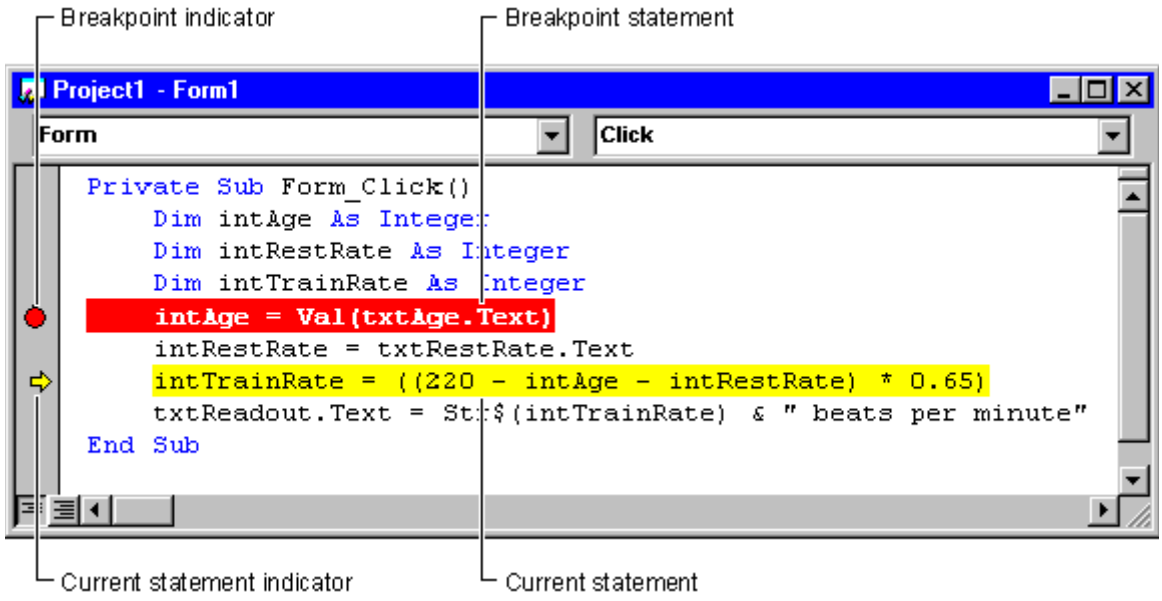
20Press F9.

33

When you set a breakpoint, Visual Basic highlights the selected line in bold, using the colors that you specified on the Editor Format tab of the Options dialog box, available from the Tools menu.

For example, Figure 13.16 shows a procedure with a breakpoint on the fifth line. In the Code window, Visual Basic indicates a breakpoint by displaying the text on that line in bold and in the colors specified for a breakpoint.

Figure 13.16 A procedure halted by a breakpoint



34

Identifying the Current Statement

In Figure 13.16, a rectangular highlight surrounds the seventh line of code. This outline indicates the *current statement*, or next statement to be executed. When the current statement also contains a breakpoint, only the rectangular outline highlights the line of code. Once the current statement moves to another line, the line with the breakpoint is displayed in bold and in color again.

To specify the color of text of the current statement

21 From the **Tools** menu, choose **Options** and click the **Editor Format** tab on the **Options** dialog box.

22 Under **Code Colors**, select **Execution Point Text**, and set the **Foreground**, **Background**, and **Indicator** colors.

35

Examining the Application at a Breakpoint

Once you reach a breakpoint and the application is halted, you can examine the application's current state. Checking results of the application is easy, because you can move the focus among the forms and modules of your application, the Code window, and the debugging windows.

A breakpoint halts the application just before executing the line that contains the breakpoint. If you want to observe what happens when the line with the breakpoint executes, you must execute at least one more statement. To do this, use Step Into or Step Over.

—42

For More Information See the section, "Running Selected Portions of Your Application," later in this chapter.

47

When you are trying to isolate a problem, remember that a statement might be indirectly at fault because it assigns an incorrect value to a variable. To examine the values of variables and properties while in break mode, use the Locals window, Quick Watch, watch expressions, or the Immediate window.

For More Information To learn how to use the Immediate window to test the values of properties and variables, see "Testing Data and Procedures with the Immediate Window," later in this chapter. To learn more about watch expressions, see "Monitoring Data with Watch Expressions."

48

Using Stop Statements

Placing a Stop statement in a procedure is an alternative to setting a breakpoint. Whenever Visual Basic encounters a Stop statement, it halts execution and switches to break mode. Although Stop statements act like breakpoints, they aren't set or cleared the same way.

Caution Be sure to remove any Stop statements before you create an .exe file. If a stand-alone Visual Basic application (.exe) encounters a Stop statement, it treats it as an End statement and terminates execution immediately, without any QueryUnload or Unload events occurring.

Remember that a Stop statement does nothing more than temporarily halt execution, while an End statement halts execution, resets variables, and returns to design time. You can always choose Continue from the Run menu to continue running the application.

For More Information See "How to Handle Errors" earlier in this chapter for an example that uses the Stop statement.

49

Running Selected Portions of Your Application

If you can identify the statement that caused an error, a single breakpoint might help you locate the problem. More often, however, you know only the general area of the code that caused the error. A breakpoint helps you isolate that problem area. You can then use Step Into and Step Over to observe the effect of each statement. If necessary, you can also skip over statements or back up by starting execution at a new line.

Step Mode	Description
Step Into	Execute the current statement and break at the next line, even if it's in another procedure.
Step Over	Execute the entire procedure called by the current line and break at the line following the current line.
Step Out	Execute the remainder of the current procedure and break at the statement following the one that called the procedure.

50

Note You must be in break mode to use these commands. They are not available at design time or run time.

51

Using Step Into

You can use Step Into to execute code one statement at a time. (This is also known as single stepping.) After stepping through each statement, you can see its effect by looking at your application's forms or the debugging windows.

To step through code one statement at a time

- From the **Debug** menu, choose **Step Into**.

  21– or –

  22Click the **Step Into** button on the **Debug** toolbar. (To display the Debug toolbar, right-click on the Visual Basic toolbar and select the **Debug** option.)

  23– or –

  24Press F8.

36

When you use Step Into to step through code one statement at a time, Visual Basic temporarily switches to run time, executes the current statement, and advances to the next statement. Then it switches back to break mode.

Note Visual Basic allows you to step into individual statements, even if they are on the same line. A line of code can contain two or more statements, separated by a colon (:). Visual Basic uses a rectangular outline to indicate which of the statements will execute next. Breakpoints apply only to the first statement of a multiple-statement line.

52

Using Step Over

Step Over is identical to Step Into, except when the current statement contains a call to a procedure. Unlike Step Into, which steps into the called procedure, Step Over executes it as a unit and then steps to the next statement in the current procedure. Suppose, for example, that the statement calls the procedure SetAlarmTime:

```
SetAlarmTime 11, 30, 0
```

53

If you choose Step Into, the Code window shows the SetAlarmTime procedure and sets the current statement to the beginning of that procedure. This is the better choice only if you want to analyze the code within SetAlarmTime.

If you use Step Over, the Code window continues to display the current procedure. Execution advances to the statement immediately after the call to SetAlarmTime, unless SetAlarmTime contains a breakpoint or a Stop statement. Use Step Over if you want to stay at the same level of code and don't need to analyze the SetAlarmTime procedure.

You can alternate freely between Step Into and Step Over. The command you use depends on which portions of code you want to analyze at any given time.

To use Step Over

- From the **Debug** menu, choose **Step Over**.

  25– or –

  26Click the **Step Over** button on the **Debug** toolbar. (To display the Debug toolbar, right-click on the Visual Basic toolbar and select the **Debug** option.)

  27– or –

  28Press SHIFT+F8.

37

Using Step Out

Step Out is similar to Step Into and Step Over, except it advances past the remainder of the code in the current procedure. If the procedure was called from another procedure, it advances to the statement immediately following the one that called the procedure.

To use Step Out

- From the **Debug** menu, choose **Step Out**.

  29– or –

  30Click the **Step Out** button on the **Debug** toolbar. (To display the Debug toolbar, right-click on the Visual Basic toolbar and select the **Debug** option.)

  31– or –

  32Press CTRL+SHIFT+F8.

38

Bypassing Sections of Code

When your application is in break mode, you can use the Run To Cursor command to select a statement further down in your code where you want execution to stop. This lets you "step over" uninteresting sections of code, such as large loops.

To use Run To Cursor

23 Put your application in break mode.

24 Place the cursor where you want to stop.

25 Press CTRL+F8.

33– or –

34From the **Debug** menu, choose **Run To Cursor**.

39

Setting the Next Statement to Be Executed

While debugging or experimenting with an application, you can use the Set Next Statement command to skip a certain section of code — for instance, a section that contains a known bug — so you can continue tracing other problems. Or you may want to return to an earlier statement to test part of the application using different values for properties or variables.

With Visual Basic, you can set a different line of code to execute next, provided it falls within the same procedure. The effect is similar to using Step Into, except Step Into executes only the next line of code in the procedure. By setting the next statement to execute, you choose which line executes next.

To set the next statement to be executed

26 In break mode, move the insertion point (cursor) to the line of code you want to execute next.

27 From the **Debug** menu, choose **Set Next Statement**.

28 To resume execution, from the **Run** menu, choose **Continue**.

35 – or –

36 From the **Debug** menu, choose **Run To Cursor**, **Step Into**, **Step Over,** or **Step Out**.

40

Showing the Next Statement to Be Executed

You can use Show Next Statement to place the cursor on the line that will execute next. This feature is convenient if you've been executing code in an error handler and aren't sure where execution will resume. Show Next Statement is available only in break mode.

To show the next statement to be executed

29 While in break mode, from the **Debug** menu, choose **Show Next Statement**.

30 To resume execution, from the **Run** menu, choose **Continue**.

37 – or –

38 From the **Debug** menu, choose **Run To Cursor**, **Step Into**, **Step Over,** or **Step Out**.

41

Monitoring the Call Stack

—46

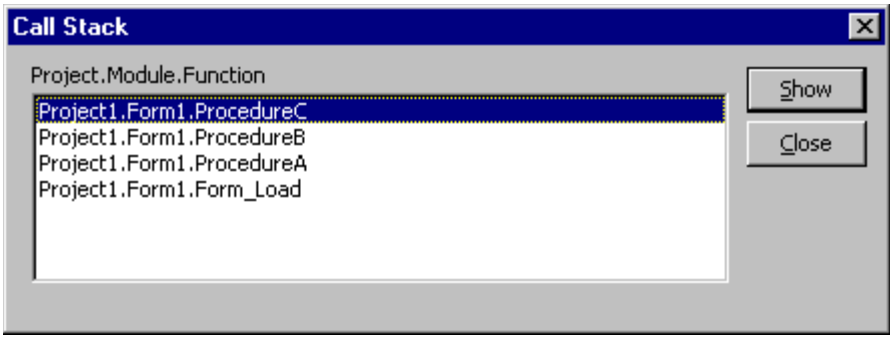
The Call Stack dialog box shows a list of all active procedure calls. *Active procedure calls* are the procedures in the application that were started but not completed.

The Call Stack dialog box helps you trace the operation of an application as it executes a series of nested procedures. For example, an event procedure can call a second procedure, which can call a third procedure — all before the event procedure that started this chain is completed. Such nested procedure calls can be difficult to follow and can complicate the debugging process. Figure 13.17 shows the Call Stack dialog box.

Note If you put the application in break mode during an idle loop, no entries appear in the Call Stack dialog box.

54

Figure 13.17 The Call Stack dialog box



42

You can display the Call Stack dialog box only when the application is in break mode.

To display the Call Stack dialog box

- From the **View** menu, choose **Call Stack**.

  39– or –

  40Click the **Call Stack** button on the **Debug** toolbar. (To display the Debug toolbar, right-click on the Visual Basic toolbar and select the **Debug** option.)

  41– or –

  42Press CTRL+L.

  43– or –

  44Click the button next to the Procedure box in the **Locals** window.

43

Tracing Nested Procedures

The Call Stack dialog box lists all the active procedure calls in a series of nested calls. It places the earliest active procedure call at the bottom of the list and adds subsequent procedure calls to the top of the list.

The information given for each procedure begins with the module or form name, followed by the name of the called procedure. Because the Call Stack dialog box doesn't indicate the variable assigned to an instance of a form, it does not distinguish between multiple instances of forms or classes. For more information on multiple instances of a form, see "Programming with Objects".

You can use the Call Stack dialog box to display the statement in a procedure that passes control of the application to the next procedure in the list.

To display the statement that calls another procedure in the Calls Stack dialog box

31 In the **Call Stack** dialog box, select the procedure call you want to display.

32 Choose the **Show** button.

45The dialog box is closed and the procedure appears in the Code window.

44

The cursor location in the Code window indicates the statement that calls the next procedure in the Call Stack dialog box. If you choose the current procedure in the Call Stack dialog box, the cursor appears at the current statement.

Checking Recursive Procedures

The Call Stack dialog box can be useful in determining whether "Out of stack space" errors are caused by recursion. *Recursion* is the ability of a routine to call itself. You can test this by adding the following code to a form in a new project:

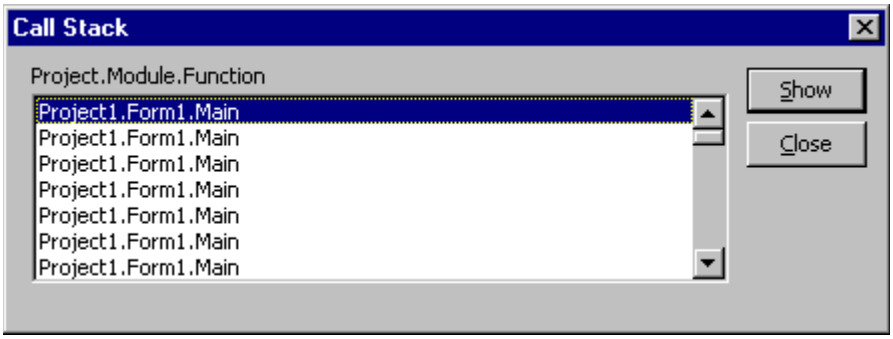
```
Sub Main()
    Static intX As Integer
    intX = intX + 1
    Main
End Sub

Private Sub Form_Click()
    Main
End Sub
```

55

Run the application, click the form, and wait for the "Out of stack space" error message. Choose the Debug button, and then choose Call Stack on the View menu. You'll see multiple calls to the Main procedure, as shown in Figure 13.18.

Figure 13.18 The Call Stack dialog box lists a recursive procedure



45

As a double check, highlight intX in the Code window, and choose Quick Watch from the Debug menu. The value for intX is the number of times the Main procedure executed before the break.

Testing Data and Procedures with the Immediate Window

Sometimes when you are debugging or experimenting with an application, you may want to execute individual procedures, evaluate expressions, or assign new values to variables or properties. You can use the Immediate window to accomplish these tasks. You evaluate expressions by printing their values in the Immediate window.

Printing Information in the Immediate Window

There are two ways to print to the Immediate window:

- Include Debug.Print statements in the application code.
- Enter Print methods directly in the Immediate window.

46

These printing techniques offer several advantages over watch expressions:

- You don't have to break execution to get feedback on how the application is performing. You can see data or other messages displayed as you run the application.
- Feedback is displayed in a separate area (the Immediate window), so it does not interfere with output that a user sees.
- Because you can save this code as part of the form, you don't have to redefine these statements the next time you work on the application.

47

Printing from Application Code

—49

The Print method sends output to the Immediate window whenever you include the Debug object prefix:

Debug.Print [*items*][;]

For example, the following statement prints the value of Salary to the Immediate window every time it is executed:

```
Debug.Print "Salary = "; Salary
```

This technique works best when there is a particular place in your application code at which the variable (in this case, Salary) is known to change. For example, you might put the previous statement in a loop that repeatedly alters Salary.

Note When you compile your application into an .exe file, Debug.Print statements are removed. Thus, if your application only uses Debug.Print statements with strings or simple variable types as arguments, it will not have any Debug.Print statements. However, Visual Basic will not strip out function calls appearing as arguments to Debug.Print. Thus, any side-effects of those functions will continue to happen in a compiled .exe file, even though the function results are not printed.

Printing from Within the Immediate Window

Once you're in break mode, you can move the focus to the Immediate window to examine data.

To examine data in the Immediate window

1. Click the **Immediate** window (if visible).

   – or –

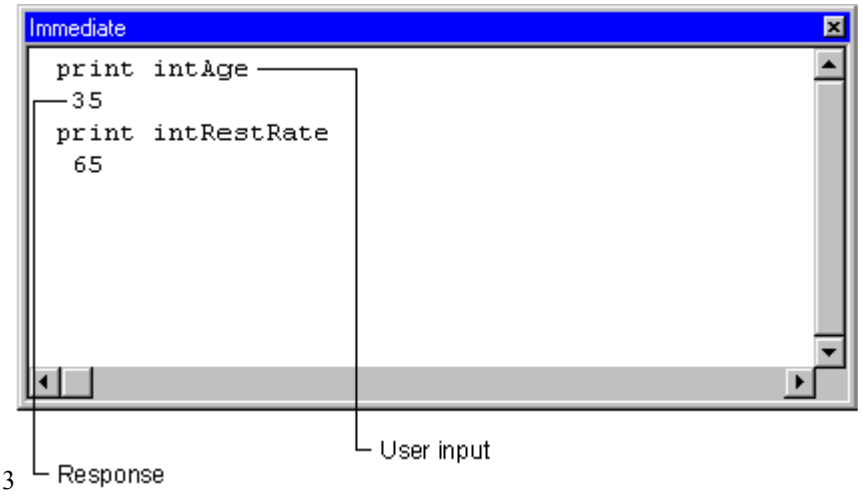
   From the **View** menu, choose **Immediate Window**.

   Once you have moved focus to the Immediate window, you then can use the Print method without the Debug object.

2. Type or paste a statement into the **Immediate** window, and then press ENTER.

   The Immediate window responds by carrying out the statement, as shown in Figure 13.19.

4Figure 13.19 Using the Print method to print to the Immediate window

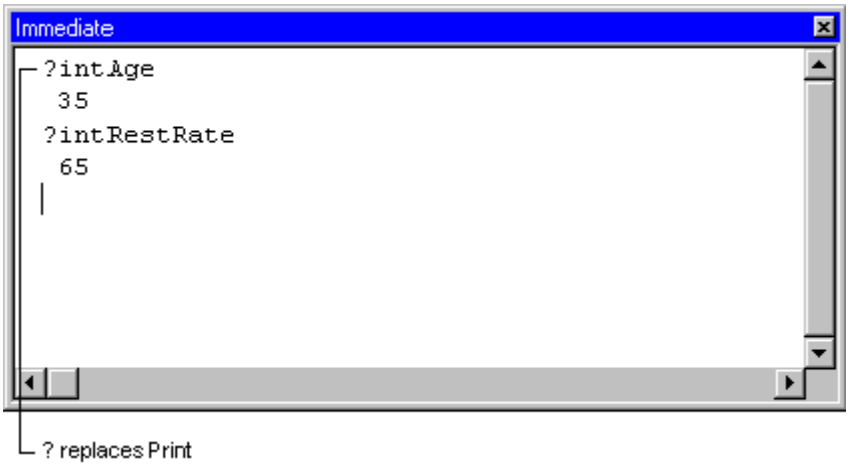


3

48

A question mark (?) is useful shorthand for the Print method. The question mark means the same as Print, and can be used in any context where Print is used. For example, the statements in Figure 13.19 could be entered as shown in Figure 13.20.

Figure 13.20 Using a question mark instead of the Print method



49

Printing Values of Properties

You can evaluate any valid expression in the Immediate window, including expressions involving properties. The currently active form or module determines the scope. If the execution halts within code that is attached to a form or class, you can refer to the properties of that form (or one of its controls) and make the reference to the form implicit with statements like the following:

—51

```
? BackColor
? Text1.Height
```

59

Assuming that Text1 is a control on the currently active form, the first statement prints the numeric value of the current form's background color to the Immediate window. The second statement prints the height of Text1.

If execution is suspended in a module or another form, you must explicitly specify the form name as follows:

```
? Form1.BackColor
? Form1.Text1.Height
```

60

Note Referencing an unloaded form in the Immediate window (or anywhere else) loads that form.

61

For More Information To learn about changing properties and values in the Immediate window, see "Assigning Values to Variables and Properties" later in this chapter.

62

Assigning Values to Variables and Properties

As you start to isolate the possible cause of an error, you may want to test the effects of particular data values. In break mode, you can set values with statements like these in the Immediate window:

```
BackColor = 255
VScroll1.Value = 100
MaxRows = 50
```

63

The first statement alters a property of the currently active form, the second alters a property of VScroll1, and the third assigns a value to a variable.

After you set the values of one or more properties and variables, you can continue execution to see the results. Or you can test the effect on procedures, as described in the next topic, "Testing Procedures with the Immediate Window."

Testing Procedures with the Immediate Window

The Immediate window evaluates any valid Visual Basic executable statement, but it doesn't accept data declarations. You can enter calls to Sub and Function procedures, however, which allows you to test the possible effect of a procedure with any given set of arguments. Simply enter a statement in the Immediate window (while in break mode) as you would in the Code window. For example:

```
X = Quadratic(2, 8, 8)
DisplayGraph 50, Arr1
Form_MouseDown 1, 0, 100, 100
```

64

When you press the Enter key, Visual Basic switches to run time to execute the statement, and then returns to break mode. At that point, you can see results and test any possible effects on variables or property values.

If Option Explicit is in effect (requiring all variable declarations to be explicit), any variables you enter in the Immediate window must already be declared within the current scope. Scope applies to procedure calls just as it does to variables. You can call any procedure within the currently active form. You can always call a procedure in a module, unless you define the procedure as Private, in which case you can call the procedure only while executing in the module.

For More Information Scope is discussed in "Introduction to Variables, Constants, and Data Types" in "Programming Fundamentals."

65

Viewing and Testing Multiple Instances of Procedures

You can use the Immediate window to run a procedure repeatedly, testing the effect of different conditions. Each separate call of the procedure is maintained as a separate instance by Visual Basic. This allows you to separately test variables and property settings in each instance of the procedure. To see how this works, open a new project and add the following code to the form module:

```
Private Sub Form_Click()
    AProcedure
End Sub

Sub AProcedure()
    Dim intX As Integer
    intX = 10
    BProcedure
End Sub

Sub BProcedure()
    Stop
End Sub
```

66

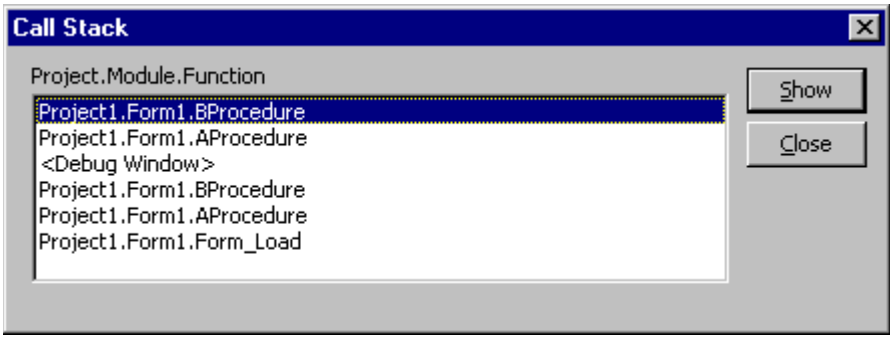
Run the application and click the form. The Stop statement puts Visual Basic into break mode and the Immediate window is displayed. Change the value of intX to 15 in the procedure "AProcedure," switch to the Immediate window, and type the following:

```
AProcedure
```

67

This calls the procedure "AProcedure" and restarts the application. If you switch to the Immediate window and run "AProcedure" again, and then open the Call Stack dialog box, you'll see a listing much like the one in Figure 13.21. Each separate run of the program is listed, separated by the [<Debug Window>] listing.

Figure 13.21 The Call Stack dialog box shows multiple instances of procedures

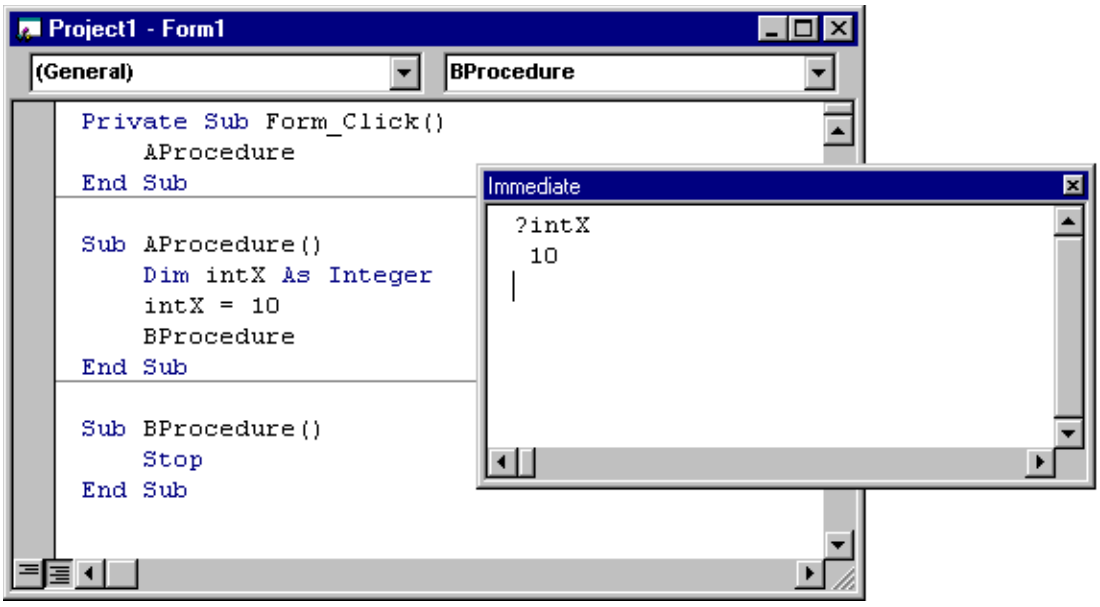


50

Visual Basic maintains a listing of the procedures executed by each command from the Immediate window. Newer listings are at the top of the list. You can use the Call Stack dialog box to select any instance of a procedure, and then print the values of variables from that procedure in the Immediate window.

For example, if you double-click the earliest instance of "AProcedure" and use the Immediate window to print the value of intX, it will return 10, as shown in Figure 13.22. If you changed the value of intX to 15 for the second run of the "AProcedure," that value is stored with the second instance of the procedure.

Figure 13.22 Printing the values of variables in the Immediate window



51

Note Although most statements are supported in the Immediate window, a control structure is valid only if it can be completely expressed on one line of code; use colons to separate the statements that make up the control structure. The following For loop is valid in the Immediate window:

```
For I = 1 To 20 : Print 2 * I : Next I
```

68

Checking Error Numbers

You can use the Immediate window to display the message associated with a specific error number. For example, enter this statement in the Immediate window:

```
error 58
```

69

Press ENTER to execute the statement. The appropriate error message is displayed, as shown in Figure 13.23.

Figure 13.23 Displaying error messages from the Immediate window

52

Tips for Using the Immediate Window

Here are some shortcuts you can use in the Immediate window:

- Once you enter a statement, you can execute it again by moving the insertion point back to that statement and pressing ENTER anywhere on the line.
- Before pressing ENTER, you can edit the current statement to alter its effects.
- You can use the mouse or the arrow keys to move around in the Immediate window. Don't press ENTER unless you are at a statement you want to execute.
- CTRL+HOME will take you to the top of the Immediate window; CTRL+END will take you to the bottom.
- The HOME and END keys move to the beginning and end of the current line.

53
70

Special Debugging Considerations

Certain events that are a common part of using Microsoft Windows can pose special problems for debugging an application. It's important to be aware of these special problems so they don't confuse or complicate the debugging process.

If you remain aware of how break mode can put events at odds with what your application expects, you can usually find solutions. In some event procedures, you may need to use Debug.Print statements to monitor values of variables or properties instead of using watch expressions or breakpoints. You may also need to change the values of variables that depend on the sequence of events. This is discussed in the following topics.

Breaking Execution During MouseDown

If you break execution during a MouseDown event procedure, you may release the mouse button or use the mouse to do any number of tasks. When you continue execution, however, the application assumes that the mouse button is still pressed down. You don't get a MouseUp event until you press the mouse button down again and then release it.

When you press the mouse button down during run time, you break execution in the MouseDown event procedure again, assuming you have a breakpoint there. In this scenario, you never get to the MouseUp event. The solution is usually to remove the breakpoint in the MouseDown procedure.

Breaking Execution During KeyDown

If you break execution during a KeyDown procedure, similar considerations apply. If you retain a breakpoint in a KeyDown procedure, you may never get a KeyUp event. (KeyDown and KeyUp are described in "Responding to Mouse and Keyboard Events.")

Breaking Execution During GotFocus or LostFocus

If you break execution during a GotFocus or LostFocus event procedure, the timing of system messages can cause inconsistent results. Use a Debug.Print statement instead of a breakpoint in GotFocus or LostFocus event procedures.

Testing and Using Command-Line Arguments

You can choose to have your application use command-line arguments, which provide data to your application at startup. The user can enter them by choosing the operating environment's Run command, and then typing arguments after the application name. You can also use command-line arguments when creating an icon for the application.

For example, suppose you create an alarm clock application. One of the techniques for setting the alarm time is to let the user type in the selected time directly. The user might enter the following string in the Run dialog box:

```
Alarm 11:00:00
```

71

The Command function returns all arguments entered after the application name (in this case, Alarm). The Alarm application has only one argument, so in the application code, you can assign this argument directly to the string that stores the selected time:

```
AlarmTime = Command
```

72

If Command returns an empty string, there are no command-line arguments. The application must either ask for the information directly or select a default action.

To test code that uses Command, you can specify sample command-line arguments from within the Visual Basic environment. The application evaluates sample command-line input the same way it does if the user types the argument.

To set sample command-line arguments

35 From the **Project** menu, choose **Properties**.

36 Click the **Make** tab on the **Project Properties** dialog box.

37 Enter the sample arguments in the **Command Line Arguments** field. (Do not type the name of the application itself.)

38 Choose **OK**.

39 Run the application.

54

Removing Debugging Information Before Compiling

If you do not want debugging statements included in the application you distribute to users, use conditional compilation to conveniently delete these statements when the Make EXE File command is used.

For example:

```
Sub Assert(Expr As Boolean, Msg As String)
    If Not Expr Then
        MsgBox Msg
    End If
End Sub

Sub AProcedure(intX As Integer)
    # If fDebug Then
        Assert intX < 10000 and intX > 0, _
        "Argument out of range"
    # End If
        ' The code can now assume the correct value.
End Sub
```

73

Because the call to the Assert procedure is conditionally compiled, it is only included in the .exe file if fDebug is set to True. When you compile the distribution version of the application, set fDebug to False. As a result, the .exe file will be as small and fast as possible.

Note Beginning with Visual Basic version 5.0, it is no longer necessary to create your own Assert procedures. The Debug.Assert statement performs the same function and is automatically stripped from the compiled code. See "Verifying Your Code with Assertions" later in this chapter for more information.

74

Verifying Your Code with Assertions

Assertions are a convenient way to test for conditions that should exist at specific points in your code. For instance, you may assume that a certain variable's value will always be between 1 and 100 within a specific code segment; an assertion will alert you only if your assumption isn't correct.

In Visual Basic, assertions take the form of a method: the Assert method of the Debug object. The Assert method takes a single argument of the type Boolean which states the condition to be evaluated. The syntax for the Assert method is as follows:

Debug.Assert(*boolean expression*)

75

A Debug.Assert statement will never appear in a compiled application, but when you're running in the design environment it causes the application to enter break mode with the line containing the statement highlighted (assuming that the expression evaluates to False). The following example shows the Debug.Assert statement:

```
Debug.Assert Trim(CustName) <> "John Doe"
```

76

In this case, if the CustName is John Doe, the application will enter break mode; otherwise the execution will continue as usual. Using Debug.Assert is similar to setting a watch with the Break When Value Is True option selected, except that it will break when the value is false.

Using Compile on Demand

Compile on Demand and Background Compile are related features that allow your application to run faster in the development environment. It's possible that using these features may hide compile errors in your code until you make an exe for your entire project. Both features are turned on by default, and they can be turned on or off on the General tab of the Options dialog box available from the Tools menu.

Compile on Demand allows your application, in the development environment, to compile code only as needed. When Compile on Demand is on and you choose Start from the Run menu (or press the F5 key), only the code necessary to start the application is compiled. Then, as you exercise more of your application's capabilities in the development environment, more code is compiled as needed.

Background Compile allows Visual Basic at run time in the development environment to continue compiling code if no other actions are occurring.

With these features turned on, some code may not be compiled when a project is run in the development environment. Then, when you choose to Make EXE file (or turn off Compile on Demand), you may see new and unexpected errors as that code is newly compiled.

There are three techniques you can use at development milestones, or any other time, to flush out any errors hidden by using Compile on Demand.

- Turn Compile on Demand off and then run the application. This forces Visual Basic to check the entire application for compile errors.
- Make an executable with your project. This will also force Visual Basic to check the entire application for compile errors.
- Choose Start With Full Compile from the Run menu.

55

Tips for Debugging

There are several ways to simplify debugging:

- When your application doesn't produce correct results, browse through the code and try to find statements that may have caused the problem. Set breakpoints at these statements and restart the application.
- When the program halts, test the values of important variables and properties. Use Quick Watch or set watch expressions to monitor these values. Use the Immediate window to examine variables and expressions.
- Select Break on All Errors on the General tab of the Options dialog box (available from the Tools menu) to determine where an error occurred. Step through your code, using watch expressions and the Locals window to monitor how values change as the code runs.
- If an error occurs in a loop, define a break expression to determine where the problem occurs. Use the Immediate window together with Set Next Statement to re-execute the loop after making corrections.
- If you determine that a variable or property is causing problems in your application, use a Debug.Assert statement to halt execution when the wrong value is assigned to the variable or property.

56

For More Information Breakpoints are described in "Using a Breakpoint to Selectively Halt Execution" earlier in this chapter. Read more about Watch expressions in "Monitoring Data with Watch Expressions." The Immediate window is discussed in "Testing Data and Procedures with the Immediate Window." See "Verifying Your Code with Assertions" for more about the Assert method of the Debug object.

77