0672321025 AppA 10/24/01 3:53 PM Page 1

APPENDIX

A

Debugging Code in Access 2002

IN THIS APPENDIX

- Setting the Correct Module Options for Maximum Debugging Power 2
- Using the Immediate Window 6
- Stopping Program Execution 10
- Debugging One Step at a Time 12
- Viewing the Order of Procedure Calls 14
- Watching Expressions During Program Execution 15
- Controlling Code with Conditional Compilation Commands 22

0672321025 AppA 10/24/01 3:53 PM Page 2

While creating an application, you can spend much time and effort trying to figure out those bugs that creep into the system. These bugs can greatly slow down the completion of the application.

This appendix discusses the Access 2002 tools that handle bugs and examine code while creating an application. Mainly because of the Visual Basic Editor (VBE), debugging changed in Access 2000 from previous versions. Although the available commands remain the same as they were in earlier versions (with new ones added as of Access 2000), the way to use the commands changed.

Setting the Correct Module Options for Maximum Debugging Power

The Access VBA environment includes the same editor as other Office products, as well as Visual Basic. Because the VBE is in its own MDI space, you must go to the editor to set the coding environment options.

To change or view the settings of the VBA environment, use the Options dialog for VBA. For example, to change the color of a code line with a syntax error in it, follow these steps:

1. Create a new database, or open an existing one.
2. Press Alt+F11 to open the Visual Basic Editor.
3. From the Tools menu, choose Options. The VBA Options dialog appears, with the following tabbed pages: Editor, Editor Format, General, and Docking.
4. On the Editor Format page, select Syntax Error Text from the Code Colors list box (see Figure A.1).

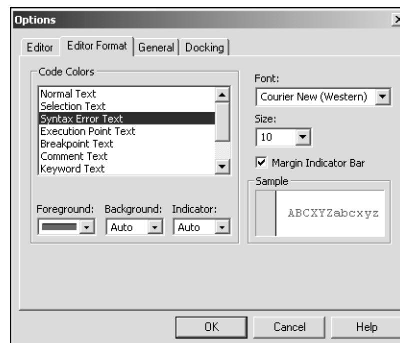

FIGURE A.1
The Editor Format page in the Options dialog contains a number of options used for debugging purposes.

Setting the color of code items is one of many ways to set up the application environment for maximum debugging power. The advantage of coloring code is that you can tell what's happening with different parts of the code. Red, for example, denotes a syntax error.

Table A.1 lists the various commands and their default color settings. The Foreground and Background columns refer to the specific code line discussed. VBA uses color along the left side of the module editor, called a *margin indicator*, to help point out various commands that have been placed in the module editor.

TABLE A.1 Default Colors for Various Code Syntax

Text Area	*Foreground*	*Background*	*Indicator*
Normal	Automatic	Automatic	Automatic
Selection	Automatic	Automatic	Automatic
Syntax Error	Light Red	Automatic	Automatic
Execution Point	Automatic	Dark Yellow	Dark Yellow
Breakpoint	White	Dark Red	Dark Red
Comment	Light Green	Automatic	Automatic
Keyword	Dark Blue	Automatic	Automatic
Identifier	Automatic	Automatic	Automatic
Bookmark	Automatic	Automatic	Cyan
Call Return	Automatic	Automatic	Light Green

NOTE

When a color is set to Automatic, Access uses the setting for the default Windows system colors.

TIP

Notice that one of the colors is for Bookmark. Bookmarks are used to tag code lines that you want to remember later for whatever reason. This feature is very useful in big chunks of code. A square with rounded corners appears in the margin indicator bar. You can use the selections on the Edit menu for Bookmark (toggle), Next Bookmark, Previous Bookmark, and Clear All Bookmarks. Bookmarks disappear when you close the database.

Other useful coding options are found on the different pages in the Options dialog. From the Editor page, you have these code settings:

- **Auto Syntax Check.** When selected, this option makes Access generate an error with a message box while you type code lines. The message box doesn't appear unless an error exists after you complete a line and press Enter. It's recommended that you leave this option set to its default (on) when starting out with VBA, and then turn it off when you are comfortable with recognizing syntax errors.
- **Require Variable Declaration.** When enabled, this option places the `Option Explicit` statement in the Declarations section of any new modules created. It doesn't affect previously created modules. It's recommended that you change this option from its default (off) to on.

TIP

If you turn on Require Variable Declaration and use `Option Explicit` in each module, you'll save countless hours of searching for misspelled variables. For more information on explicit versus implicit variable declarations, refer to the book's Chapter 2, "Coding in Access 2002 with VBA."

- **Auto List Members.** Selecting this option causes a list of possible options (that is, properties and methods) to appear when you're building a statement in code.
- **Auto Quick Info.** With this option on, function syntax appears below the code line on which you're working. This will reflect the function, statement, or object you're now typing. This also includes user-defined procedures.
- **Auto Data Tips.** By setting this option to true, you will see the value of the variable you have the cursor over when in a break of program execution.

TIP

The preceding three options have the dumb name of *IntelliSense* but are very hot. Auto Data Tips in particular are great because you don't have to highlight values and click Quick Watch. All you do is place the cursor over the variable name to examine it (see Figure A.2).

- **Auto Indent.** This option indents code automatically to enhance readability for debugging and maintaining code. It's recommended that you leave this option set to its default (on).

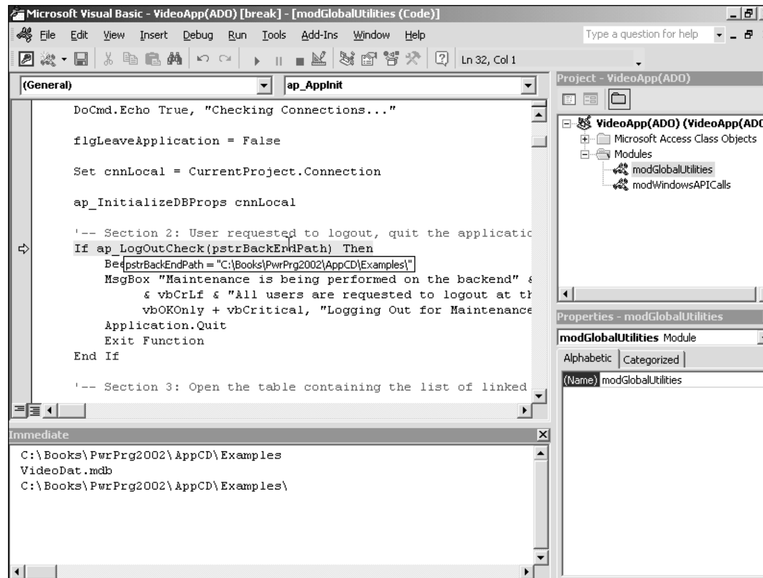

FIGURE A.2
Looking at variables such as `pstrBackEndPath` *is a breeze with the Auto Data Tips option.*

The rest of the commands that affect the debugging environment are on the General page (see Figure A.3):

- **Notify Before State Loss.** Enable this option to specify whether you want Access to tell you when your variables will be reset at the module level when a running project is halted.
- **Error Trapping.** These three options let you choose when you want Access to break on errors that can occur in your code:

Break on All Errors	Has Access break on all errors that occur on the line where it occurs. This includes whether error handlers are active or the code is in a class module.
Break in Class Module	Has Access break in class modules. If this option isn't specified, the line that calls a property or method of the class module will display the error.
Break on Unhandled Errors	Causes Access to break on errors that don't have an error handler already in use.

- **Compile on Demand.** This option, enabled by default, keeps Access from compiling the entire potential call tree when you start up a form. VBA compiles only as you call various functions.

- **Background Compile.** By selecting this option, Access will compile any uncompiled code when Access's processes are idle.

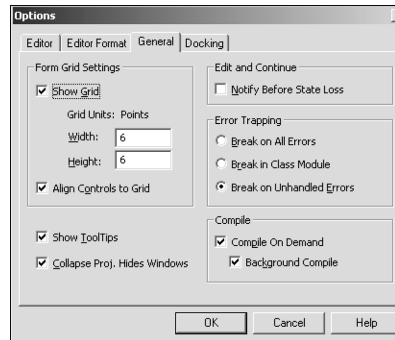

FIGURE WEB A.3
The options on the right side of this page can affect the debugging process.

NOTE

To ensure that no compile errors are lurking in obscure forms, choose Compile *Project_Name* from the Debug menu while in the module editor before distributing a system. Also, because Access doesn't have to compile the code during runtime while in production, the application performs better. You should also select Save from the File menu to save the project after compiling.

Using the Immediate Window

The VBE Immediate window has a number of features that allow you to follow code and to examine expressions in a number of different ways. By using the Immediate window, you can print data and reassign values to variables and Access objects, which come in handy as you debug your application. To bring the Immediate window up while in the VBE, open the View menu and choose Immediate window.

TIP

Press Ctrl+G anywhere in your application to bring up the VBE Immediate window.

0672321025 AppA 10/24/01 3:53 PM Page 7

Printing Data to the Immediate Window From Your Application

You can print information to the Immediate window by using the `Print` method available on the `Debug` object. When a code line uses the `Debug.Print` method, it prints the data to the Immediate window, even if the window isn't open. The next time you open the window, the text will be there. Figure A.4 shows how to include the `Debug.Print` method in your code, as well as the output in the Immediate window.

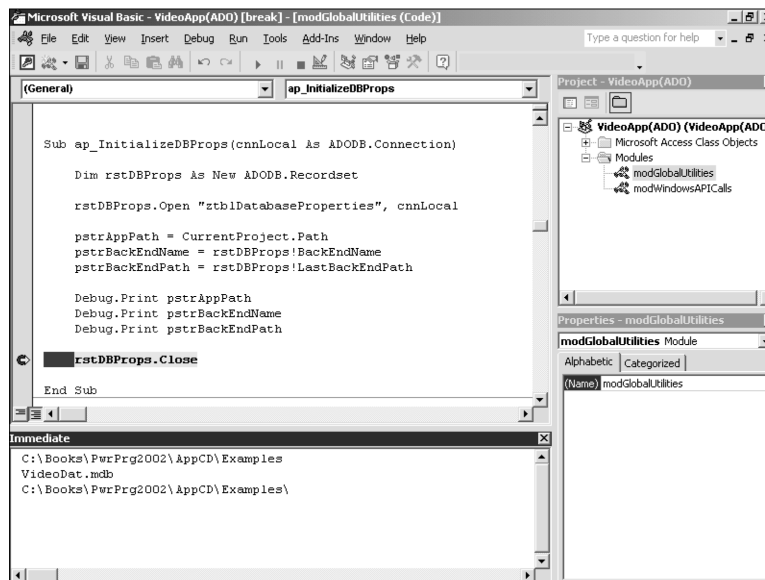

FIGURE A.4

The `Debug.Print` method is useful for keeping track of expressions in your application.

NOTE

Although you can't see the code lines that contain the `Debug.Print` statement if the Immediate window isn't open, Access still has to take the time to parse the command at runtime and print to the object. If you don't want to include these code lines, use the conditional compilation directives mentioned later in the section "Controlling Code with Conditional Compilation Commands."

Displaying Data While in the Immediate Window

By using the `?` (Print) statement, you can display all types of expressions while in an application. Simply place a `Stop` statement or breakpoint in your application, open the Immediate window, and then use the `?` statement to print the information you're interested in. (`Stop` statements and breakpoints are discussed in detail later in the section "Setting Breakpoints and Using the `Stop` Statement.")

Suppose that you're running a routine that runs through a recordset and checks a value for one of the fields. By displaying the value of that field to the Immediate window, you could see what the value is during execution for each record.

The following are some examples of different data types that you can display while in the Immediate window:

Object Type	*Syntax*
Control on a form	`? Forms!FormName!ControlName`
Property of a control	`? Forms!FormName!ControlName.PropertyName`
Variable	`? VariableName`
Variable in an expression	`? VariableName * 20`

Assigning Values to Variables and Objects in the Immediate Window

The Immediate window also lets you assign values and perform commands right from the window. You also can perform actions such as closing recordsets manually.

TIP

Through the Immediate window, you can assign a new value to a variable, and then use the Run menu's Set Next Statement command to place the next line of execution just before that value is used in the program. You can then use the Step commands to walk through the code and view the results. The Step commands are discussed in greater detail later in the section "Debugging One Step at a Time."

Look at the following table. You can use similar syntax for assigning values:

Object Type	*Syntax*
Control on a form	`Forms!FormName!ControlName = 1`
Property of a control	`Forms!FormName!ControlName.PropertyName = "New Name"`

Object Type	*Syntax*
Variable	`VariableName = 3`
Variable in an expression	`VariableName = VariableName * 20`

Running Code from the Immediate Window

One nice thing about the Immediate window is that it gives you the capability to test routines without requiring a complete framework around the routine. You can call routines straight from the Immediate window by simply typing the necessary syntax in the window. The syntax for calling the two types of procedures is as follows:

- For a subroutine, the syntax is

  ```
  SubName arg1,arg2...
  ```

  or

  ```
  Call SubName (arg1,arg2...)
  ```

- For a function, the syntax is as follows if you want to print the function's return value:

  ```
  ? FunctionName(arg1,arg2...)
  ```

  Or use this syntax if you're just running the function and don't care about the results:

  ```
  FunctionName arg1,arg2...
  ```

NOTE

The functionality of running functions on a line by themselves was new as of Access 95. Before that, functions couldn't be called on a line by themselves. This also was true when calling functions at runtime.

Figure A.5 shows an example of calling a subroutine and a function.

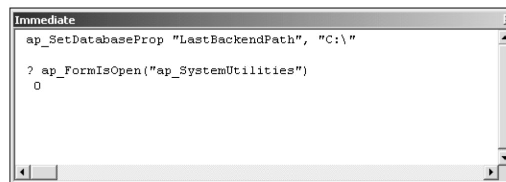

FIGURE A.5
Use the Immediate window to test subroutines and functions on-the-fly.

Stopping Program Execution

Bringing up the Immediate window during runtime is no problem. You can try to pause an application programmatically with either a `Stop` statement or a breakpoint. When placed in code, both methods will halt the execution of a piece of code and bring up the module editor, with the code line containing the `Stop` statement or breakpoint highlighted. You can have as many breakpoints and `Stop` statements in your code as you want.

Figure A.6 shows a `Stop` statement and a breakpoint set in the `ap_AppInit()` function, which is in the `modGlobalUtilities` module.

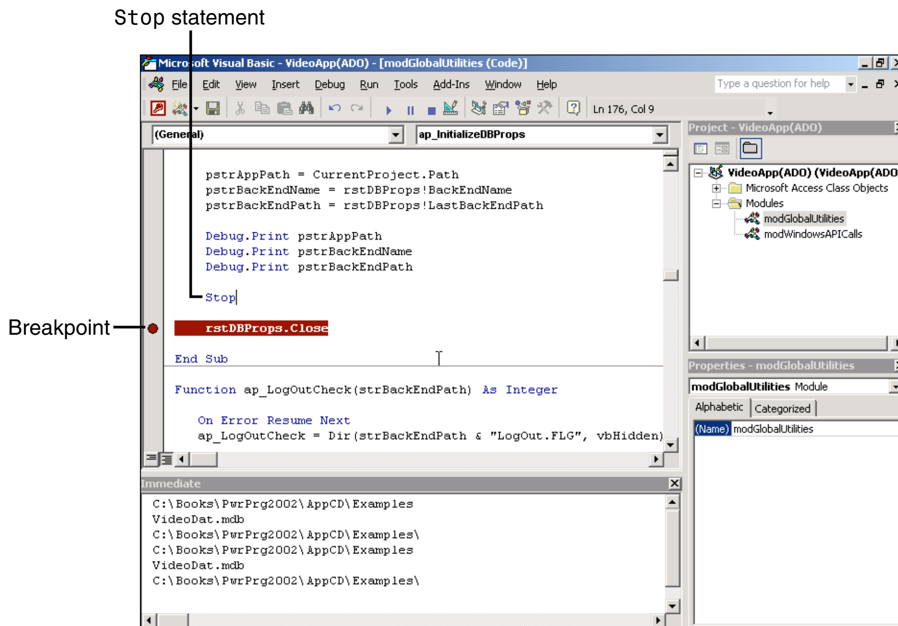

FIGURE A.6

Breakpoints and `Stop` statements are two ways to stop execution at a specific point in code.

NOTE

You can't place `Stop` statements or breakpoints in comments, variable declarations (the `Dim` statement), blank lines, line labels, or line numbers.

Using the `Stop` Statement

To place a `Stop` statement in a code line, simply type the command.

CAUTION

`Stop` statements, if saved with the module, remain in the code until you remove them. Remember to remove all `Stop` statements from your code and forms modules before distributing your application for production. To find all occurrences of `Stop` statements, follow these steps:

1. Open a module.
2. From the Edit menu, choose Replace.
3. Type **`Stop`** in the Find What text box. (Leave the Replace With text box blank.)
4. Select Current Project in the Search section.
5. Click the Find Next button.
6. Click the Replace button if the text found is a legitimate `Stop` statement. If the text is part of another line of code, such as comments or a text string, skip this step.
7. Repeat steps 5 and 6 until you see the message `The specified region has been searched`.

Although it's not required, if you turn on Find Whole Word Only in the Replace dialog, you'll be less likely to get bogus occurrences of the string.

Using Breakpoints

The alternative to a `Stop` statement is a breakpoint. Breakpoints are useful because when the database is closed, they go away. As a result, you don't have to worry about breakpoints left in your code when you distribute the application.

To set a breakpoint, first highlight the line on which you want to stop code execution. Then you can toggle (set or unset) a breakpoint in one of a number of ways:

- Choose the Breakpoint toolbar button.
- Right-click, and then choose Breakpoint from the Toggle menu.
- Press F9.
- Click in the left margin next to the code line you want to set your breakpoint on. No highlight is required.
- From the Debug menu, choose Toggle Breakpoint.

If the breakpoint didn't exist before, you'll see it at this time, highlighted in whatever color you set the Breakpoint Text color code to be. (The default color is dark red for the background, white for the foreground.) You'll also see a red dot if the margin indicator bar is on. To unset a breakpoint, follow the same steps that were performed to set it. (For information on how to set the breakpoint text color, refer to the earlier section "Setting the Correct Module Options for Maximum Debugging Power.")

To remove all breakpoints in an application, open the Debug menu and choose Clear All Breakpoints while in the VBE.

TIP

Inserting a breakpoint in your code can really help you track down an error in your code when a variable is returning a `Null` value when it shouldn't be. You could place a breakpoint just after the point in the code where the variable is being assigned, and examine the environment with some of the other commands listed later in the section "Watching Expressions During Program Execution."

Using `Debug.Assert`

The `Debug` object's `Assert` method stops execution of an application based on criteria. The syntax looks like this:

```
Debug.Assert booleanexpression
```

The assertion stops execution with the last statement executed, the `Debug.Assert` line.

Debugging One Step at a Time

To work through some debugging situations, you need to be able to walk a program line by line while it's executing. VBA provides four `Step` commands to accomplish this: `Step Into`, `Step Over`, `Step Out`, and `Run to Cursor`. You can use these commands after a program halts execution by choosing the appropriate command from the Debug menu.

NOTE

All four `Step` commands skip over the same type of code lines that don't allow break-points: comments, declaring variables (the `Dim` statement), blank lines, and line labels or line numbers.

Stepping into Code Line by Line

The Step Into command steps through code lines one by one. When a code line has a call to another procedure, the editor then follows the code into it. This includes user-created functions and subroutines, but not intrinsic VBA functions such as `Date()` and `Mid()`. Calls into DLLs and OLE servers are also omitted.

First, halt program execution by pressing Ctrl+Break when the routine is executing, by setting a breakpoint, or by using a `Stop` statement. Then you can step through the code line by line, with the program pausing on each line, by either choosing the Step Into button of the Visual Basic toolbar, opening the Debug menu and then choosing Step Into, or pressing F8.

Stepping Through Code with Step Over

Similar to the Step Into command, the Step Over command takes you line by line through program execution. The difference comes when you're on a function or subroutine being called from within the original routine. Step Over is useful when you have a number of thoroughly debugged routines that can therefore be skipped.

To use the Step Over command, simply press Shift+F8. When you use Step Over, rather than drop into the new routine, the program will

1. Execute the new routine without displaying the code.
2. Begin displaying the code line by line following the procedure call.

Bailing Out of a Routine with Step Out

Sometimes when the going gets too tough, it's best just to bail out of the current routine and resume with the calling routine, in the line of code that follows the call.

The Step Out command allows you to leave a routine that might be messed up. You also can use Step Out if you don't need to bother with a particular routine, but accidentally went into it with the Step Into command.

You can use the Step Out command by pressing Ctrl+Shift+F8.

Skipping Tested Code with Run to Cursor

Think of when you're debugging the beginning of a routine. After the section of the code is tested, you find that you need to jump to the end of the routine. The method for doing this is Run to Cursor.

To perform a Run to Cursor from within the halted program, place the cursor where you want Access to execute the code to without stopping. Then press Ctrl+F8. The execution highlight will now be on the line of code in which you placed the cursor.

0672321025 AppA 10/24/01 3:53 PM Page 14

When you want the program to continue with regular execution, press F5.

One more debug/code option is Set Next Statement, which sets the code execution to whatever code line you select. You can set the cursor on a line and then choose Set Next Statement from the Debug menu, or right-click a code line and select Set Next Statement. This is useful if you are quickly testing a particular piece of code and are feeding it different values “manually” by setting it in the Immediate, Locals window, or Watches windows. You can continue executing the same line(s) of code to test different values.

NOTE

Unlike the Step options, Set Next Statement does exactly what it says—it goes to the statement you select without executing any code at all. Keep this in mind when using Set Next Statement.

Another option is Show Next Statement, which brings you back to the code line that is executed next (the line that’s highlighted yellow). This is useful if you are stepping through the code and paging through different modules and code windows to do your debugging, and need to get back to the executing code. You can find Show Next Statement on the Debug and right-click menus.

Viewing the Order of Procedure Calls

Another necessary debugging tool that Access provides is the capability to view procedure calls. You can get to the Calls dialog through a couple of different routes, but this section focuses on the Immediate window.

One reason for viewing the procedure calls is that sometimes you need to verify how a routine was called. This is especially true if you have more than one way to get into a routine.

To view the procedure call stack from the VBE, follow these steps:

1. Open the `modGlobalUtilities` module.
2. Go to the `ap_AppInit()` function.
3. Place a `Stop` statement in the first line of code following the function declaration. Then close the module.
4. Execute the function by running the AutoExec macro. When the code reaches the `Stop` statement, the module window opens with that code displayed.
5. By using the Step Into command (F8), step through the code until you reach the line that reads

   ```
   pstrAppPath = CurrentProject.Path
   ```

6. Press Shift+F8 to step over this line of code. You'll now be on the line of code that reads

   ```
   pstrBackendName = ap_GetDatabaseProp("BackEndName")
   ```

7. Press F8 to step into the `ap_GetDatabaseProp()` function.
8. From the View menu, choose Call Stack. (You also can open the Call Stack dialog by pressing Ctrl+L or by using the Call Stack dialog toolbar button.) Your dialog should resemble Figure A.7.

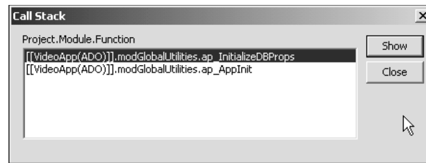

FIGURE A.7
Use the Call Stack dialog to verify which functions are being called correctly.

The Call Stack dialog lists the procedure calls from top to bottom, with the most recent on top. To see any of the procedures, highlight the procedure name in the Project.Module.Function list, and then click Show. The editor then displays the chosen routine.

Watching Expressions During Program Execution

One very necessary feature in programming is the capability to watch expressions throughout execution and have the debugger react accordingly. In the past, you might have done this by using a message box to display the value of an expression (see Figure A.8).

You also can use the `Debug.Print` method all over the code. However, a better way to keep an eye on different expressions is to use the Locals and Watches windows. You can use these windows to see how your variables are doing and—more importantly—*what* they're doing.

Keeping in Touch with the Locals

By using the Locals window, you can view the local variables not only for the current routine you are in, but also for the module level and for the Data Access Objects variable properties.

Figure A.9 illustrates the variables assigned in the declaration section for the `modGlobalUtilities` module. (They're actually global-type variables, so the term *local* in this sense means where they're declared.) Figure A.9 also shows the variables declared in `ap_AppInit`, including ADO variables, which at this point are collapsed. This will also work on ADO variables displayed on the Watches window.

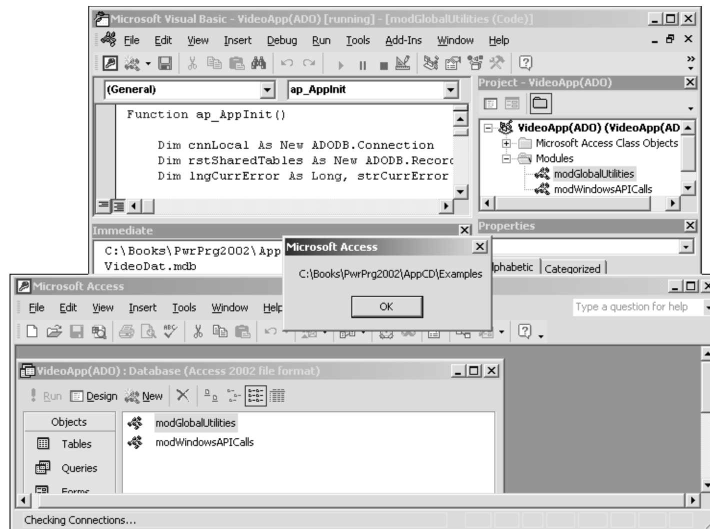

FIGURE A.8

Using a message box to display expressions is a common debugging method.

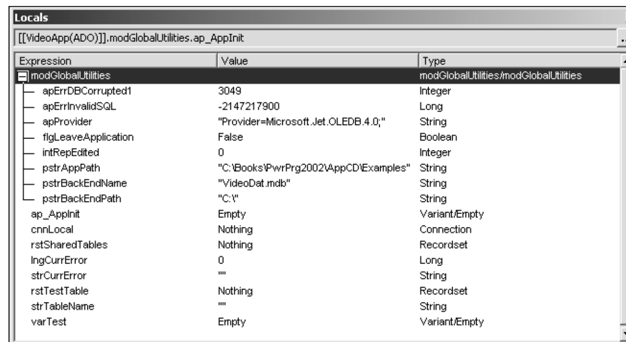

FIGURE A.9

To see the current setting of collapsed routines or variables, click the variable or routine name.

The Locals window displays three columns:

- **Expression.** As with the earlier section “Printing Data to the Immediate Window,” an expression can be anything from a variable to a control value on a form.

0672321025 AppA 10/24/01 3:53 PM Page 17

- **Value.** This column shows the displayed value. If the expression isn't declared in the given procedure and module, the Value column in the Watches window will say `Expression not defined in context`.
- **Type.** This column displays the expression's data type.

Another quick way to view expressions when the program is executing through the Step commands is by using a Quick Watch (discussed next). Other commands included in the Immediate window and the Edit Watch dialog are discussed later.

TIP

A good way to learn about objects' properties, what they contain when, and to view a hierarchy, is to set watches on them and expand. You also can directly change many values right in the Watches/Locals windows by clicking and typing.

Taking a Quick Look with the Quick Watch Dialog

The Quick Watch dialog is a convenient way to examine an expression with a click (or two) of a button. This is very useful when you have a routine that just doesn't seem to be acting the way it should. You can look at the return value for the routine anywhere in the program that you think would be pertinent. To invoke Quick Watch and view the answer for the routine `ap_LogOutCheck`, for example, follow these steps:

1. Open the `ap_AppInit()` function in the `modGlobalUtilities` module.
2. Place a `Stop` statement in the first line of code after the line of code that reads

   ```
   pstrBackEndPath = ap_GetDatabaseProp("LastBackEndPath")
   ```
3. Open the Immediate window and enter **`Ap_AppInit()`**. The editor appears with the `Stop` statement highlighted.
4. Place the cursor on the function call `ap_LogOutCheck(pstrBackEndPath)`.
5. From the Debug menu, choose Quick Watch.

Your screen should now look very similar to Figure A.10.

In the Quick Watch dialog, you're one click away from adding the expression to the Watches window.

Adding and Viewing Expressions in the Watches Window

If you're examining a variable, such as `pstrAppPath`, you can add it to the Watches window by clicking the Add button in the Quick Watch dialog. After you do this, you can then see the

0672321025 AppA 10/24/01 3:53 PM Page 18

Watches window added to the other windows in the VBE (see Figure A.11). Remember that you can bring the VBE to the front if it isn't already open by pressing Ctrl+G.

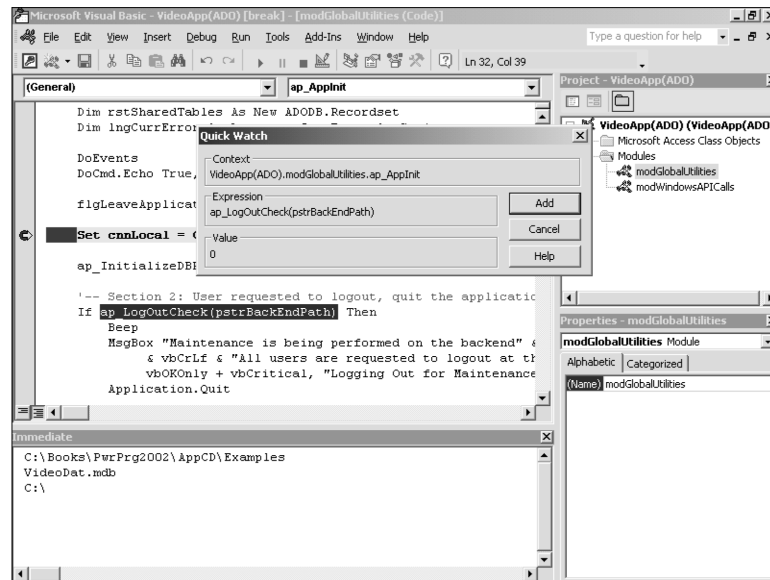

FIGURE A.10

Quick Watch is a quick way to view an expression while an application is executing, without putting additional commands in the code.

The Watches window displays four columns, three of which are the same as for the Locals page:

- **Expression.** As in the earlier section "Printing Data to the Immediate Window," an expression can be anything from a variable to a control value on a form.
- **Value.** This column shows the displayed value. If the expression isn't declared in the given procedure and module, this column will say `Expression not defined in context`. If you open the Watches window when an application isn't running, or the Module and Procedure of the watch are out-of-scope, this column will say `Out of Context`.
- **Type.** This column displays the expression's data type.
- **Context.** This is the scope in which you want to view the expression. This can be for a particular routine in a module or be throughout the whole database.

0672321025 AppA 10/24/01 3:53 PM Page 19

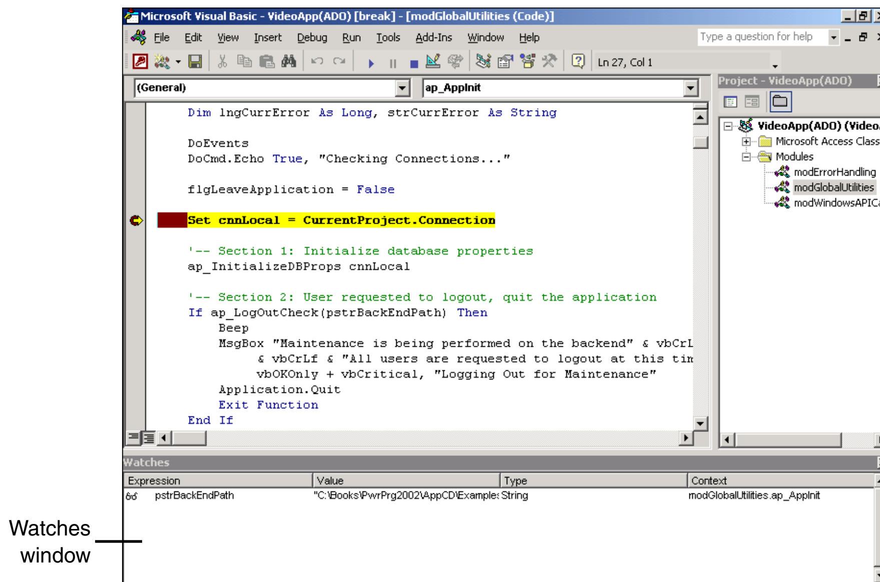

FIGURE A.11

Add expressions to the Watches window so you can follow the status without having to manually display them all the time or hunt them down on the Locals page.

NOTE

When you add an expression to the Watches window through the Quick Watch dialog or the menu, the Context column defaults to the current procedure and module (context is where the code is now running).

Another way to add an expression to the Watches window (without using the Quick Watch dialog) is by choosing Add Watch from the Debug menu. You can use this method anywhere in the VBE.

By watching an expression, you're using the Watches window in its simplest form, without any of the additional options mentioned in the next section. Because Access allows you to keep the VBE and Watches window open wherever the application is, you can watch expressions at any time.

After adding an expression and following it throughout the application, you'll soon find that you have uses for the other capabilities in the Watches window. These include having the program break when the value of an expression is true or even changes.

Setting Break Conditions and Editing Expressions

Add a couple more expressions to the Watches window. Wherever you are in the VBE, follow these steps:

1. From the Debug menu, choose Add Watch.
2. Type **`flgLeaveApplication`** in the Expression column.
3. Click OK in the Add Watch dialog.
4. Repeat steps 1 through 3 for the `lngCurrError` and `pstrBackEndPath` variables.

If you're still on the same line as the previous example, the Watches window should look like Figure A.12.

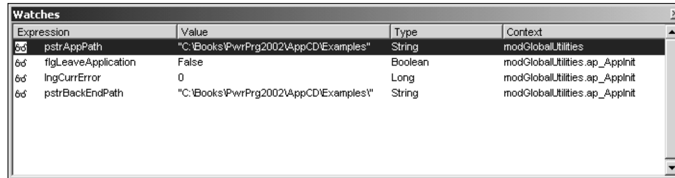

FIGURE A.12
You can view as many expressions in the Watches window as necessary.

TIP

There are two reasons for placing variables that can be found in the Locals window in the Watches window:

- You don't have to search through all the existing locals to monitor the specific variables you're interested in.
- You can set the watch variables to be different types, which is discussed following this tip.

You can now change the type of the watch variables and control execution of the program through them. To halt the program on a certain condition, you must edit the expressions. To edit the `flgLeaveApplication` expression in the Watches window, highlight the line, and choose Edit Watch from the Debug menu. You're now in the Edit Watch dialog (see Figure A.13), which consists of the following:

0672321025 AppA 10/24/01 3:53 PM Page 21

- **Expression.** This text box contains the name of the variable/expression to watch.
- **Context Group.** With this set of controls, you can specify the context—in other words, in which specific procedure/module you want to watch the expressions. You can also specify (All Procedures)/(All Modules) accordingly if you want to follow a variable that's declared global, or at module level. If the expression isn't declared in the given procedure and module, the Value field in the Watches window will say `Expression not defined in context`. For more information on variable scoping, refer to Chapter 2.
- **Watch Type.** This is where you specify the type of watch variable: Watch Expression, Break When Value Is True, or Break When Value Changes. Each option is useful, depending on the circumstances.

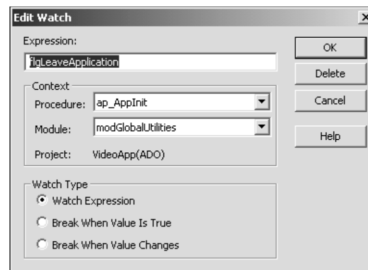

FIGURE A.13
Modifying how a variable is watched is easy with the Edit Watch dialog.

Change the Watch Type of `flgLeaveApplication` to Break When Value Is True by clicking the radio button next to the label. As the code is executing, this will allow you to know whether the variable has been set to true without walking through the code line by line.

Next, highlight the `lngCurrError` expression in the Watches window; then choose Edit Watch from the Debug menu. Now change the Watch Type to Break When Value Changes. This way, if an error occurs, the program will halt right away.

In the Local and Watches windows, you also can change variable values by clicking a value and changing it.

In Figure A.14, you can see that the watch expression (at the bottom) is denoted by a pair of glasses. The Watch Type choice Break When Value Is True is denoted by a hand with a piece of paper in it (the top expression). Finally, the Watch Type choice Break When Value Changes is denoted with an icon of a hand holding a triangle (the middle expression).

0672321025 AppA 10/24/01 3:53 PM Page 22

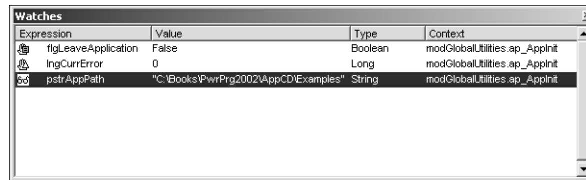

FIGURE A.14
You not only can watch expressions, but also can set them to cause the application to halt under certain conditions.

Controlling Code with Conditional Compilation Commands

Although Access doesn't create true executables, it does compile the VBA code to a level lower than the Visual Basic language the developer works with. This compiled code is called *pseudocode* (also known as *P-code*). When compiling P-code, certain commands, such as comments, are stripped out to optimize the code.

By using conditional compilation directives, you can include or exclude code sections in the final compiled code. This is different from using just `If..Endif` to skip over parts of code while it's running. Access literally strips the section of code that's surrounded by the directives. This means that you must know which lines of code you won't need at runtime.

One way to use the conditional compilation directives would be to follow the example in the sample application. World Wide Video could have three versions of the system:

- A video kiosk, so customers can view current and upcoming movies
- A turn-key retail system, so salespeople can rent movies and sell merchandise
- An administrative system for management, with accounting and reporting capabilities

By having the conditional compilation directives in the code, you can use the same code base but include only the code used for the particular system, thus saving memory.

Although this last example could work, it's not necessarily practical because the number of forms, reports, and other objects necessary to run each version, and the number of places to put the compiler directives, would be enormous. A more practical example—and how most developers use conditional compilation directives—is to use them for debugging purposes. There's only one conditional compilation directive: `#If..#ElseIf..#Else..#End If`.

Figure A.15 shows code that prints three variables to the Immediate window if the `ccDebug` variable is set to `True`.

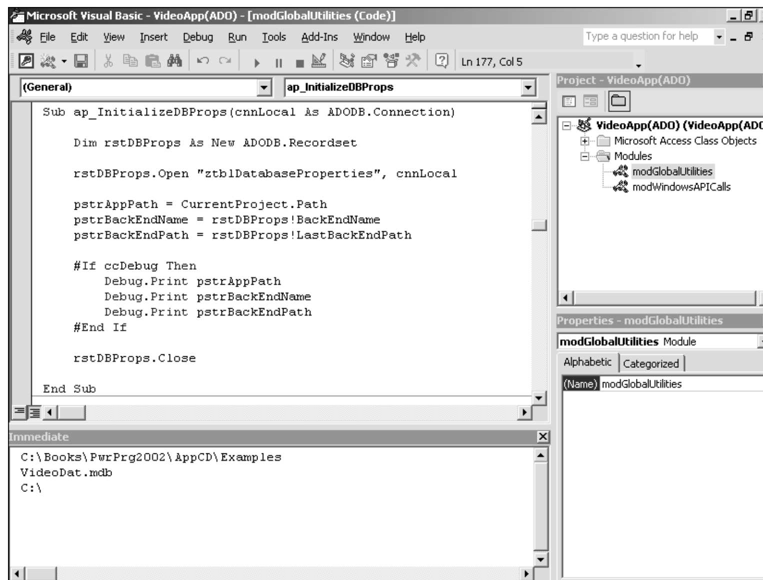

FIGURE A.15

Conditional compilation commands are useful for excluding debugging commands from distributed applications.

`ccDebug` is a *conditional compiler constant*, a special constant that can be used only with the conditional compiler directive. You can declare conditional compiler constants in two ways:

- Through the user interface on the *Project_Name* Properties sheet (choose *Project_Name* Properties from the Tools menu in the VBE). You can see `ccDebug` in the Conditional Compilation Arguments text box in Figure A.16.

FIGURE A.16

You can specify a conditional compiler constant through the Project Properties sheet.

NOTE

Conditional compiler constants declared through the UI are scoped globally. This type of constant can be only of data type `Integer`, and can't be a variable, standard constant, or even another conditional compiler constant.

- Through the `#Const` command. Conditional compiler constants are visible only at the module level and—unlike those declared through the UI—can be different data types. The `#Const` statement looks a lot like the standard `Const` statement. Here's an example using the `ccDebug` constant:

```
#Const ccDebug = 1
```

NOTE

A conditional compiler constant declared with `#Const` must be a literal, any data type, or another conditional compiler constant. It can't be a variable or standard constant.