MARCO CANTÙ'S DELPHI POWER BOOK

"**The Delphi Power Book**" is a collection of advanced essays, articles, and left-out chapters from past editions of Marco's books. The material is slowly getting a book status. Some of the text might not have been updated jet to the latest versions of Delphi.

A printer version might become available in the future. For now it is available only as an e-book in PDF format.

This chapter "**Debugging Delphi Programs**" has been last updated on April 9, 2002 and is being released in Version 0.1.

DEBUGGING DELPHI PROGRAMS

Delphi's integrated debugger

Other debugging techniques

Windows message flow

Compiled program information

Looking into memory

Once you compile a program in Delphi and run it, you may think you're finished, but not all of your problems may be solved. Programs can have run-time errors, or they may not work as you planned. When this happens, you will need to discover what has gone wrong and how to correct it. Fortunately, many options and tools are available for exploring the behavior of a Windows application.

Delphi includes an integrated debugger and several other tools to let you monitor the result of a compilation process in different ways. This chapter provides an overview of all these topics, demonstrating the key ideas with simple examples. The first part of the chapter covers Delphi's integrated debugger and various features Delphi provides for run-time debugging. Then, I'll describe some other debugging techniques and discuss how you can monitor the flow of messages in your application. The final section describes how you can examine the status of the memory used by a program.

Using the Integrated Debugger

As I've mentioned before, when you run a program from within the Delphi environment, the internal debugger actually executes the program. (You can change this behavior by disabling the Integrated Debugger option in the Debugger Options dialog box.) Most of the Run commands relate to the debugger. Some of these commands are available also in the Debug submenu of the Editor's shortcut menu.

When a program is running in the debugger, clicking on the Pause button on the SpeedBar suspends execution. Once a program is suspended, clicking on the Step Over button executes the program step by step. You can also run a program step by step from the beginning, by pressing the Step Over button while in design mode. Consider, however, that Windows applications are message-driven, so there is really no way to execute an application step by step all the way, as you can do with a DOS application. For this reason, the most common way to debug a Delphi application (or any other Windows application) is to set some *breakpoints* in the portions of the code you want to debug.

When a program has been stopped in the debugger, you can continue executing it by using the Run command. This will stop the program on the next breakpoint. As an alternative, you can monitor the execution more closely by tracing the program. You can use the Step Over command (the F8 key) to execute the next line of code, or the Trace Into command (F7) to delve into the source code of a function or method (that is, to execute the code of the subroutines step by step and to execute the code of the subroutines called by the subroutines, and so on). Delphi highlights the line that is about to be executed with a different color and a small arrow-shaped icon, so that you can see what your program is doing.

A third option, Trace to Next Source Line (Shift + F7), will move control to the next line of the source code of your program being executed, regardless of the control flow. This source code line might be the following line of the source code (as in a Step Over command), a line inside a function called by your code (as in a Trace Into command), or a line of code inside an event handler or a callback function of the program activated by the system. If you want to monitor the effect of the execution of a given line of code, you can also move to that position and call Run to Cursor (F4). The program will run until it reaches that line, so this is similar to setting a temporary breakpoint. Finally, the new Delphi 5 command Run until Return (Shift + F8) executes the method or function until it terminates. This is handy to use when you accidentally trace into a function you are not interested in debugging.

Debugging Libraries (and ActiveX Controls)

You can also use the integrated debugger to debug a DLL or any other kind of library (such as an ActiveX control). Simply open the library's Delphi project, choose the Run ➢ Parameters menu command, and enter the name of the Host Application. (This option is available only if the current project's target is a library.) Now when you press the Run button (or the F9 key), Delphi will start the main executable file, which will then load the library. If you set a breakpoint within the library source code, execution will stop in the library code, as expected.

Similarly, you can use this capability to debug an ActiveForm. Simply enter the full path name of the Web browser as the Host application, for example `C:\Program Files\Microsoft Internet\Iexplore.exe`; then enter the full path name of the test HTML file as the Run parameter. To make this work, you should also use the Run ➢ Register ActiveX Server menu command. Once the ActiveX is registered, the Web browser will use that version and not another version available in its OCX cache.

Debug Information

To debug a Delphi program, you must add debug information to your compiled code. Delphi does this by default, but if it has been turned off, you can turn it back on through the Project Options dialog box. As shown in Figure 18.1, the Compiler page includes a Debugging section with four check boxes:

- **Debug Information** adds to each unit a map of executable addresses and corresponding source code line numbers. This increases the size of the DCU file but does not affect the size or speed of the executable program. (The linker doesn't include this information when it builds the EXE file, unless you explicitly request TD32 debug information, which is technically in a different format.)
- **Local Symbols** adds debug information about all the local identifiers, the names and types of symbols in the implementation portion of the unit.
- **Reference Info** adds reference information about the symbols defined in a module to allow the Project Inspector (or Object Browser) to display them. If the suboption **Definitions Only** is not checked, the reference information tracks where each symbol is used, enabling the cross-references in the Project Explorer.
- **Assertions** allows you to add assertions, code that will stop your program if a specific test condition fails. Unlike exceptions or other error-detecting code, assertions can be removed from your program automatically; just deselect this option. I'll discuss assertions later in this chapter. After changing this setting, you have to rebuild your project to add or remove the assertion code from your application.
- **Use Debug DCUs** to link the debug version of the DCU files of the VCL in your program. In practice, this option adds the Debug DCU path (specified in General page of the Debugger Options) to the Search path (specified in Directories/Conditionals page of the Project Options dialog box).

The integrated debugger uses this information while debugging a program. Debug information is not attached to the executable file unless you set the Include TD32 Debug Info option in the Linker page of the Project Options dialog box. You should add debug information to your executable file only if you plan to use an external debugger, such as Borland's Turbo Debugger for Windows (TD32). Do not include it if you plan to use only the integrated debugger, and remember to remove it from the executable file that you ship.

Remote Debugging

A feature first introduced in Delphi 4 is remote debugging. This technique allows you to debug a program that is running on a different computer, typically a server. To activate remote debugging, you must first install the remote debugger client on the target machine. Then you should start the remote debugging client, with the command `borrdg.exe -listen`, eventually starting it as a service on Windows NT.

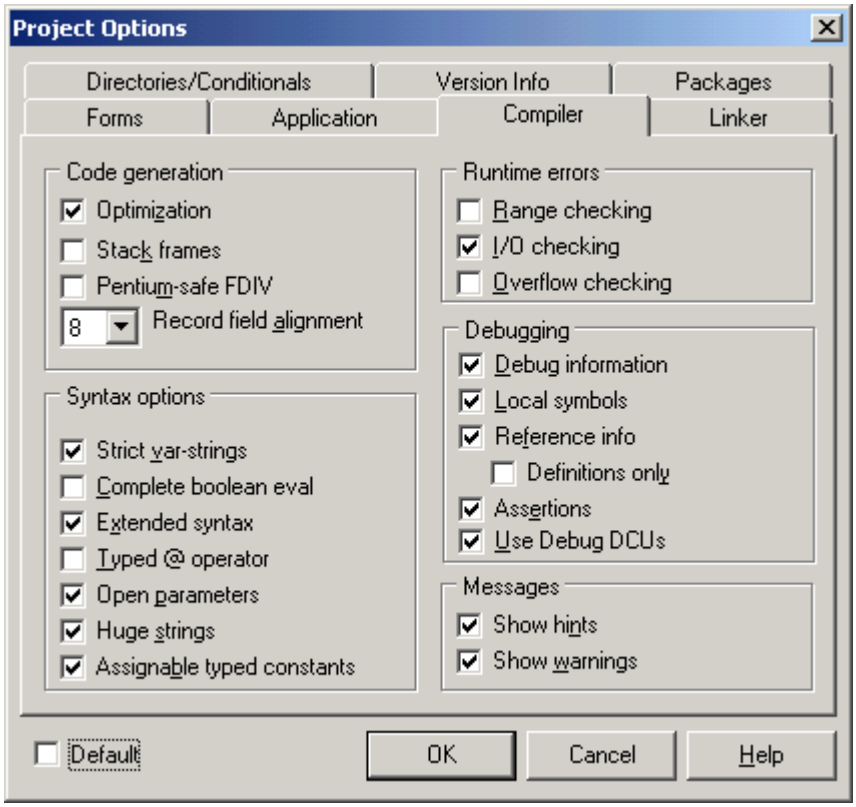


Figure 18.1: Use the Compiler page of the Project Options dialog box to include debug information in a compiled unit.

Now you should compile your program including remote debug symbols in the Linker page of the Project Options dialog box. You can also set the Output directory to the remote machine in the Directories Conditional page of the same dialog, so that you don't have to manually copy the program and the companion RMS file on the remote computer each time you recompile the program. Finally, set the remote path and project in Run ➢ Parameters, filling the Remote box with the machine name or its IP address.

When everything is properly set up, you'll be able to use the Delphi integrated debugger to debug the program running on the remote computer. You'll be able to set breakpoints and perform all of the standard debugging operations as usual.

Attach to Process

New in Delphi 5 is the Attach to Process feature, available via the Run command. This feature allows you to start debugging a program that is already running in the system. For example, you might want to attach the debugger to a process that has just displayed an exception to understand what has gone wrong.

When you select the Attach to Process command, you'll see a list of running processes. If you choose a Delphi program for which you have the source code, you'll be back to the traditional debugging situation. If you choose another program for which you don't have the source code, you'll only be able to trace its execution at the assembly level, using the CPU window but not the source code editor.

Using Breakpoints

There are several breakpoint types in Delphi:

- **Source breakpoints** and **address breakpoints** are similar, as they halt execution when the processor is about to execute the instructions at a specific address.
- **Data breakpoints** halt execution when the value at a given location changes.
- **Module load breakpoints** halt execution when a specific code module loads.

As the name implies, a breakpoint, when reached, is supposed to stop the program execution. In Delphi 5, breakpoints can do more than just stop—each breakpoint can have any of several actions associated with it. These actions can be the traditional break action, the display of a fixed string or a calculated expression in the message log, or the activation or deactivation of other groups of breakpoints. The Breakpoint List window (see Figure 18.2) shows this extra information, along with the description of the breakpoints of the entire program. The figure is taken from a simple example, BreakP, I'm using to illustrate some of the features of breakpoints in Delphi.

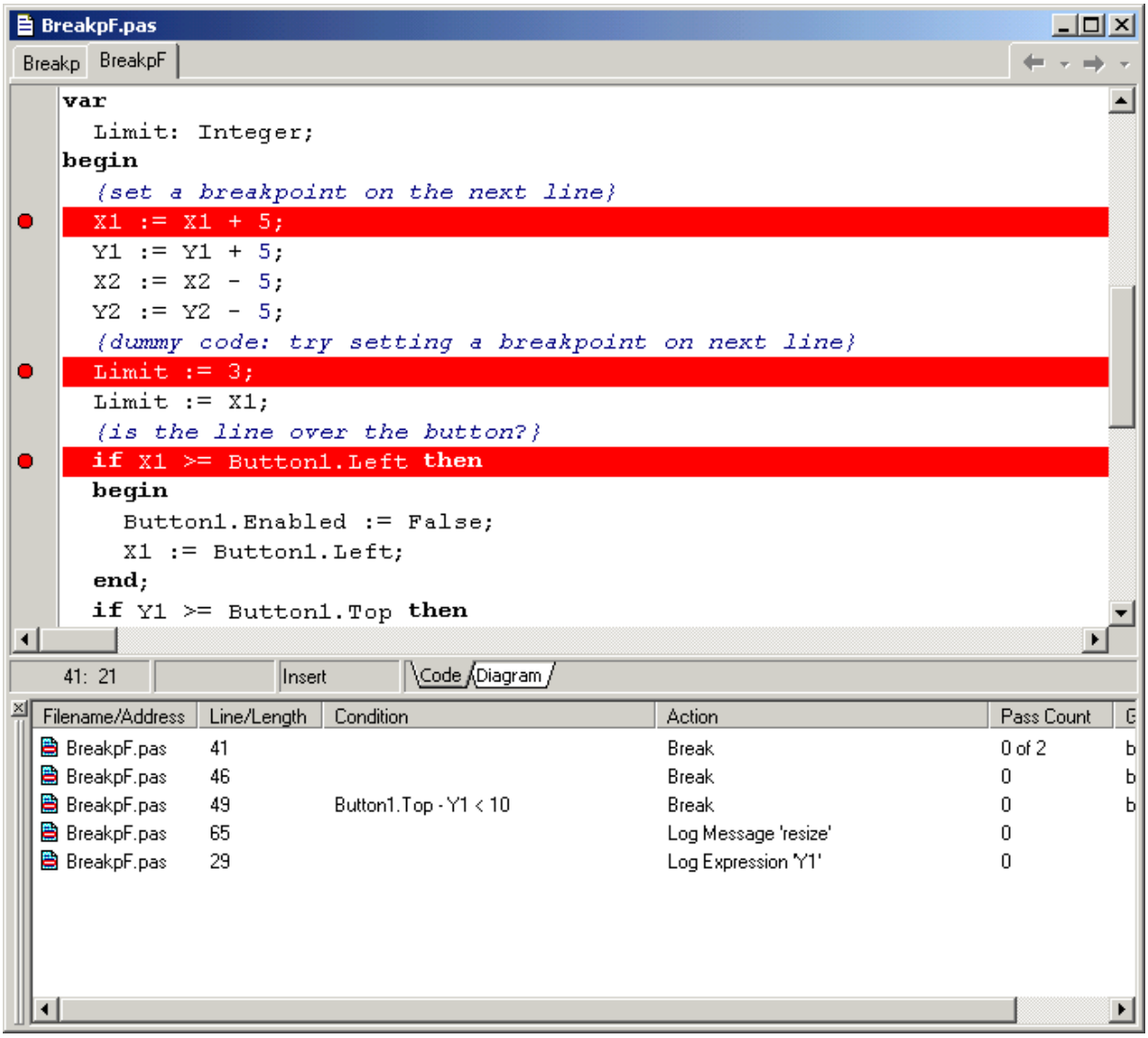


Figure 18.2: The Breakpoint List window, with a conditional breakpoint, docked on the bottom of the editor

Another new feature is that you can assign breakpoints to groups. You can then enable or disable all the breakpoints of a group at once, either using a direct command (in the shortcut menu of the Breakpoint List window) or as the automatic effect of a breakpoint action.

Source Breakpoints

If you want to break upon execution of statements in your source code, you'll use a source breakpoint, which is the most common type of breakpoint. You can create a source breakpoint by clicking in the gutter region of the code editor window, by right-clicking on a specific line in a source file and choosing the Toggle Breakpoint command from the shortcut menu, or by using the Add Source Breakpoint dialog box. (You can display this

dialog box by choosing the Run ➢ Add Breakpoint ➢ Source Breakpoint menu command, by choosing Add ➢ Source Breakpoint from the Breakpoint List window's local menu, or by pressing F5.)

When you create a new source breakpoint (using any of these techniques), an icon will appear in the margin to the left of the code, and the source line will display in a different color. However, you can't set a valid breakpoint on just any line of the source code. A source breakpoint is valid only if Delphi generated executable code for that source line. This means you can't specify a breakpoint for a comment, declaration, compiler directive line, or any other statement that's not executable. If you've compiled a program at least once (with debugging information enabled), small dots in the frame to the left of the gutter area indicate source lines where you can place a valid breakpoint. Although you can set a breakpoint at an invalid location, Delphi will alert you when you run the program, and it will mark the invalid breakpoint with a different icon and color.

Also note that because Delphi uses an optimizing compiler, it will not generate any executable code for *unreachable* lines of source code in your program nor for any other lines that have no effect on the program logic. If you create an invalid source breakpoint and then execute the program step by step, the debugger will skip the line, because that code doesn't exist in the optimized version of the compiled program. There is an example of an invalid breakpoint in the BreakpF unit of the BreakP program.

Once you've set a valid source code breakpoint, you can change some of its properties. The Source Breakpoint Properties dialog box is available from the Breakpoint list window by right-clicking on the icon in the editor gutter. In this window (see Figure 18.3), you can set a condition for the breakpoint, indicate its pass count, and assign it to a group. You can also click the Advanced button to display an extended version of the window, offering the breakpoint action features introduced in Delphi 5, which I'll discuss in the next section.

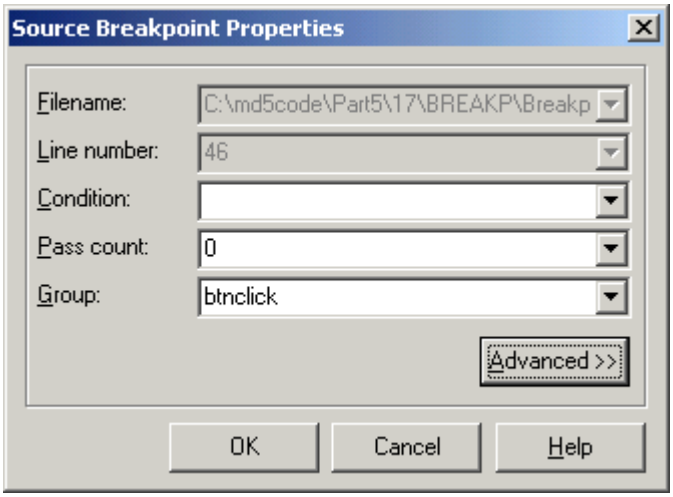


Figure 18.3: The standard portion of the Source Breakpoint Properties dialog box

In Figure 18.3, you can see the definition of a *conditional* breakpoint, used to stop the program only when a given expression is true. For example, we might want to stop the `Button1Click` method's execution of the BreakP example only when the lines (indicated by the Y1 variable) have moved near the button, using this condition:

```
Button1.Top - Y1 < 10
```

The condition is also added to the Breakpoint List window, as you can see in Figure 8.2. Now you can run the program and click the button a number of times. The breakpoint is ignored until the condition is met (when `Button1.Top - Y1` is less than 10). Only after you click several times will the program actually stop in the debugger.

If you know how many times a line of code should be allowed to execute before the debugger should stop, you can set a value for the Pass Count. As soon as the debugger reaches the breakpoint, it will increase the count until the specified number of passes has been reached. The Breakpoint List window will indicate the status (see the first line of Figure 18.2), displaying "2 of 5" if the program has executed the line only twice after you've set a pass count of 5.

The Keep Existing Breakpoint check box of the Breakpoint Properties window is used when you want to duplicate an existing point by moving the source code line it refers to. If the check box is not selected, the existing breakpoint is moved; if it is selected, a new breakpoint is created. This provides a sort of cloning mechanism if you want to create a new breakpoint based on an existing one.

Finally, notice that in Delphi 5 all of the information displayed by the Breakpoint List window is available also in a fly-by hint displayed as you move the mouse over the breakpoint icon in the editor gutter, as shown in Figure 18.4.

Figure 18.4: The new fly-by hint for breakpoints

> ☞The breakpoints you add to a program are saved in the project desktop file (`*.dsk`) if Desktop Saving is enabled. By saving this information, you'll be able to reopen the project and restart your debugging session. You can also keep your existing breakpoints, at least the complex ones, for future use: simply disable them, and save the project desktop settings instead of removing the breakpoints.

Breakpoint Actions

As I've already mentioned, Delphi 5 makes breakpoints a little more flexible. Besides simply breaking the program execution, they can perform other actions, as shown in the extended version of the Breakpoint Properties window (accessible by pressing the Advanced button), illustrated in Figure 18.5.

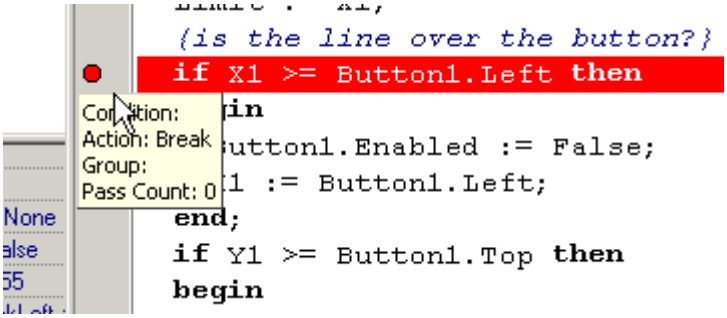


Figure 18.5: The advanced version of the Breakpoint Properties window

If you disable the Break check box, the program won't stop when the debugger reaches that line of code. You can ask for an interruption the following time the breakpoint is reached or simply send a message or the result of an expression to the Event log windows (described later), as shown in the illustration. You can also enable or disable a group of breakpoints only when a conditional breakpoint is met.

You might wonder when logging a message is better than effectively breaking the program. There are some interesting cases, as the BreakP program demonstrates. One of these message log options is used in the handler of the `OnResize` event of the form. If you drag the form border, in fact, this event will fire multiple times in a row, stopping the debugger each time.

This problem is even more obvious with the `OnPaint` event handler. If the editor window and the form overlap, you'll enter an endless series of breakpoints. Each time the form is repainted, the breakpoint stops the program, moving the editor window in front of the form and causing the form to be repainted again, which stops the program on the same breakpoint—over and over again. You could try to position the editor window and the form of your program so that they do not overlap. A more comprehensive solution is to use a conditional breakpoint, set a pass count, or log a message with the value you are interested in, such as the `Y1` variable in the case depicted in Figure 18.5.

Also Windows focus change messages are very difficult to debug with a breakpoint, because switching to the debugger for the breakpoint will force the debugged program to lose focus. You can't work around this by repositioning windows to not overlap, as for an `OnPaint` event; your choices are to use message logging or remote debugging. Remote debugging is an excellent tool for "nonintrusive" debugging of tricky state problems like painting or focus change messages.

Address Breakpoints

If you don't have the source code for a given function or procedure, you'll want to create an address breakpoint to halt execution at that point. However, without the source

code for Delphi to use in calculating the address where you wish to pause, you'll need to use some technique to determine the address of the code in question. You can create an address breakpoint either directly in the CPU view (which we'll discuss later in this chapter) or indirectly (once you have the address) using the Add Address Breakpoint dialog box.

To create an address breakpoint from the CPU view, you'll simply click in the gutter region of the Disassembly Pane, next to the instruction where you want to pause, or right-click on an instruction and select Toggle Breakpoint from the local menu. Once you've created an address breakpoint in the source code editor, you can modify its properties by right-clicking on the breakpoint icon (not on the instruction) and choosing Breakpoint Properties from the local menu.

To create an address breakpoint indirectly, you'll need to determine the address of the instruction where you want to pause execution. If you're going to pause execution of the program, but the source code isn't part of your project (as with standard VCL methods), you need to capture the address of a function or procedure that's already been compiled.

One way to determine the addresses of an object's methods is to use the Debug Inspector for that object at run time. We'll examine the Debug Inspector in more detail later in this chapter. For now, we'll just consider how you can obtain the address of a method for which you don't have the source code. For example, if you want to break when the user clicks on a button, but before entering the button's event handler code, you can locate the `TButton.Click` method. To do this, right-click on the button's declaration in the form's class declaration, and choose Debug ➢ Inspect. In the Debug Inspector window for the button, click on the Methods tab, scroll to the very end of the method list, and locate the `StdCtrls.TButton.Click` method.

Next to the name of the method, you'll see a hexadecimal address in parentheses. Copy this address (including the "$" prefix) and then create a new address breakpoint using that value. When you resume running the program and click the button, the CPU View window will appear, displaying the disassembled instructions for the `TButton.Click` method.

> ☞Unless you have the Standard edition of Delphi, you have VCL source code. You should know that one of its key uses is in debugging your applications. You can include the library source code in your program and use the debugger to trace its execution. Of course, you need to be bold enough and have enough free time to delve into the intricacies of the VCL source code. But when nothing else seems to work, this might be the only solution. To include the library source code, simply add the name of the directory with the VCL source code (by default, `Source\VCL` under your Delphi directory) to the Search Path combo box of the Directories/Conditional page of the Project

Options dialog box. An alternative is to link the Debug DCUs, as described earlier in this chapter. Then rebuild the whole program and start debugging. When you reach statements containing method or property calls, you can trace the program and see the VCL code executed line after line. Naturally, you can do any common debugging operation with the VCL code that you can do with your own code.

Data Breakpoints

Data breakpoints are dramatically different from source and address breakpoints in that they monitor a memory address for changes in value. It doesn't matter where the code resides that changes the data at that memory location; the debugger will pause immediately and display the execution point. The display will be either in the source code editor window, if the source for that code is part of the project, or the CPU View window if the source isn't available.

You can set data breakpoints two ways. The first technique involves pausing execution using a source breakpoint at a point in the source code where the identifier you want to monitor is in scope. With the program paused, you can enter the name of the identifier directly into the Address field of the Add Data Breakpoint dialog box, and Delphi will calculate the variable's address for you.

The second way to set a data breakpoint is to first create a watch for the variable, as discussed later in this chapter. Then, you'll pause execution by creating a source breakpoint at a location where the identifier is in scope. When you display the Watch List window, right-click on the watch for this identifier, and choose Break when Changed from the local menu. With either of these methods, any change in the variable's value will pause the program and display the current execution point.

Module Load Breakpoints

If you want to break when a specific code module loads, you'll create a module load breakpoint. You can create a module load breakpoint by choosing Run ➢ Add Breakpoint ➢ Module Load Breakpoint and then selecting the EXE or DLL you wish to monitor. When that module loads, Delphi will suspend the program's execution and highlight the execution point, either in the source editor window or in the CPU View window.

An easier way to achieve the same effect is to open the Modules window, run the program to a source breakpoint that occurs after the loading of all the modules, and then select the modules whose loading you wish break on. You can set a module load breakpoint for a given module by either right-clicking on the module and selecting Break

on Load from the local menu or clicking in the gutter region of the module list in the Modules window.

Debugger Views

While you are debugging a program, there are many windows (or views) you can open to monitor the program execution and its status. Most of these windows are quite intuitive, so I'll just introduce them briefly, suggesting a few hints and tips. To activate them, use the Debug submenu of the View menu of the Delphi IDE.

The Call Stack

While you're tracing a program, you can see the sequence of subroutine calls currently on the stack. Call stack information is particularly useful when you have many nested calls or when you use the Debug DCUs of the VCL. This second example is shown in Figure 18.6 for a simple `OnClick` event handler, which is called after many internal VCL calls.

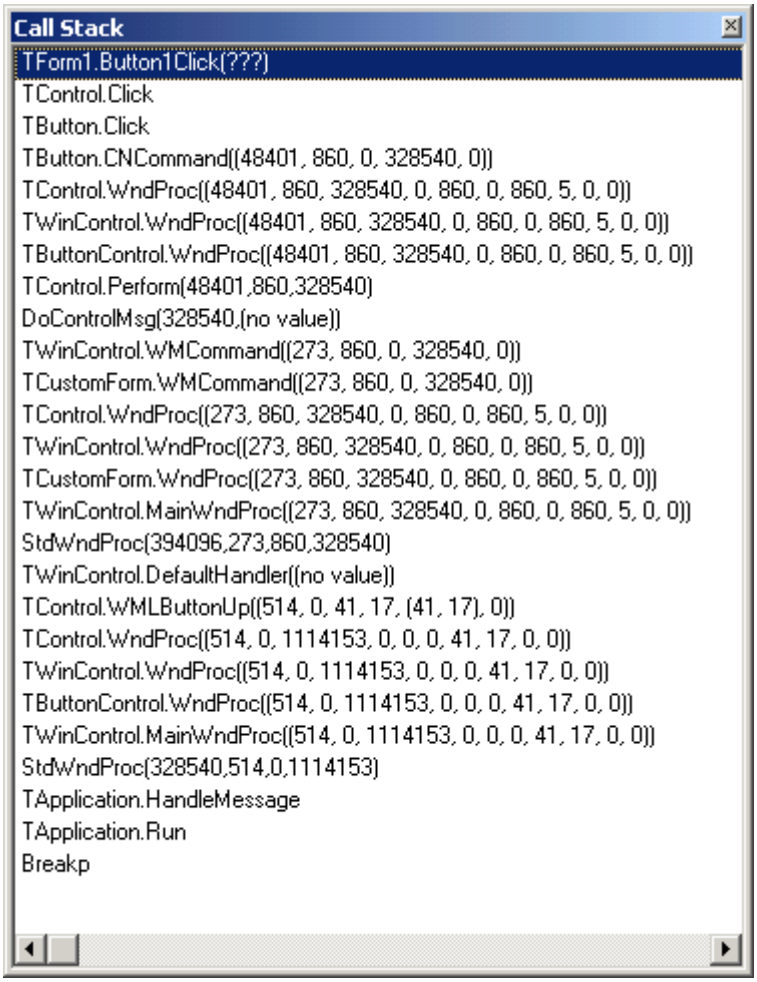


Figure 18.6: The Call stack window when a button is clicked (with Debug DCUs enabled)

The Call Stack window shows the names of the methods on the stack and the parameters passed to each function call. The top of the window lists the last function called by your program, followed by the function that called it, and so on. In the figure, you can see that the `Button1Click` event handler of `TForm1` is called by the `Click` method of the `TControl` class, which is called by the same method of the derived class `TButton`, which in turn is activated by the `WndProc` message handling method. There are multiple calls to this `WndProc` method because it is redefined in many VCL classes.

> 🕮 If you are interested, you can find a complete technical description of the steps from a Windows message to a Delphi event handler in the *Delphi Developer's Handbook* (Sybex, 1998).

Inspecting Values

When a program is stopped in the debugger, you can inspect the value of any identifier (variables, objects, components, properties, and so on) that's accessible from the current execution point (that is, only the identifiers visible in the current scope). There are many ways to accomplish this: using the fly-by debugger hints, using the Evaluate/Modify dialog box, adding a watch to the Watch List, or using the Local Variables window or the Debug Inspector.

The Fly-By Debugger Hints

When Delphi 3 introduced *fly-by evaluation hints,* this feature immediately became one of the most common ways to inspect values at run time. While a program is stopped in the debugger, you can simply move the mouse over a variable, object, property, field, or any other identifier indicating a value, and you'll immediately get a hint window showing the current value of the identifier, as you can see in Figure 18.7.

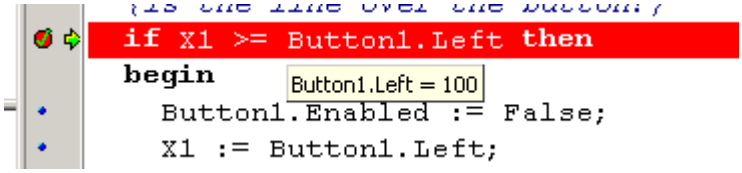


Figure 18.7: One of the more useful debugging features: fly-by evaluation hints

For simple variables, such as `X1` or `Y1` in the BreakP example, and for object properties (as in the figure), fly-by hints simply show the corresponding value, which is easy to understand. But what is the value of an object, such as `Form1` or `Button1`? Past versions of Delphi used a minimalist approach, displaying only the list of its private fields. Delphi 5 (and 6) show the entire set of properties of the object, as you can see in Figure 18.8. This is an improvement, but I think that using a Debug Inspector (see following sections) makes the status of the object more readable.

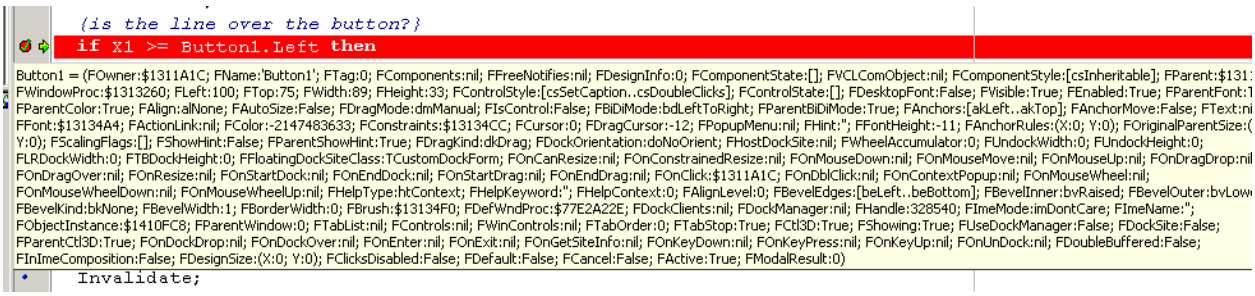


Figure 18.8: The fly-by evaluation hint for an object in Delphi 6

Remember that you can see the value of a variable when the program is stopped in the debugger but not when it is running. Additionally, you can inspect only the variables that are visible in the current scope, because they must exist in order for you to see them!

The Evaluate/Modify Window

The Evaluate/Modify dialog box can still be used to see the value of a complex expression and to modify the value of a variable or property. The easiest way to open this dialog box is to select the variable in the code editor and then choose Evaluate/Modify from the editor's shortcut menu (or press Ctrl+F7). Long selections are not automatically used, so to select a long expression, it's best to copy it from the editor and paste it into the dialog box. In Delphi 5, you can now also drag a variable or an entire expression from the source code editor to the Evaluate/Modify dialog box (see Figure 18.9).

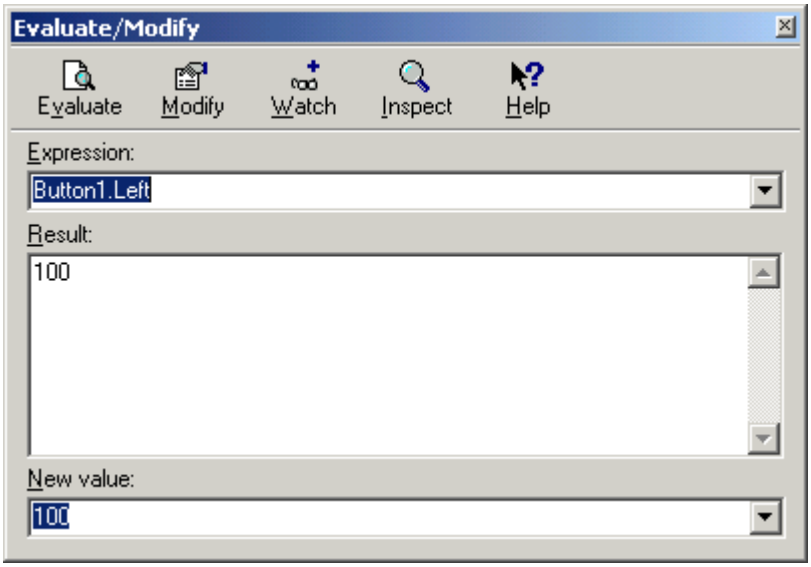


Figure 18.9: The Evaluate/Modify dialog box can be used to inspect—and change—the value of a variable. You can now also drag an expression from the editor into this window

The Watch List Window

When you want to test the value of a group of variables over and over, using the fly-by hints can become a little tedious. As an alternative, you can set some *watches* (entries in a list of variables you're interested in monitoring) for variables, properties, or components. For example, you might set a watch for each of the values used in the BreakP example's `Button1Click` method, which is called each time the user clicks on the button. I've added a number of watches to see the values of the most relevant

variables and properties involved in this method, as you can see in Figure 18.10. As mentioned earlier, you can also use this window as a starting point to set data breakpoints.

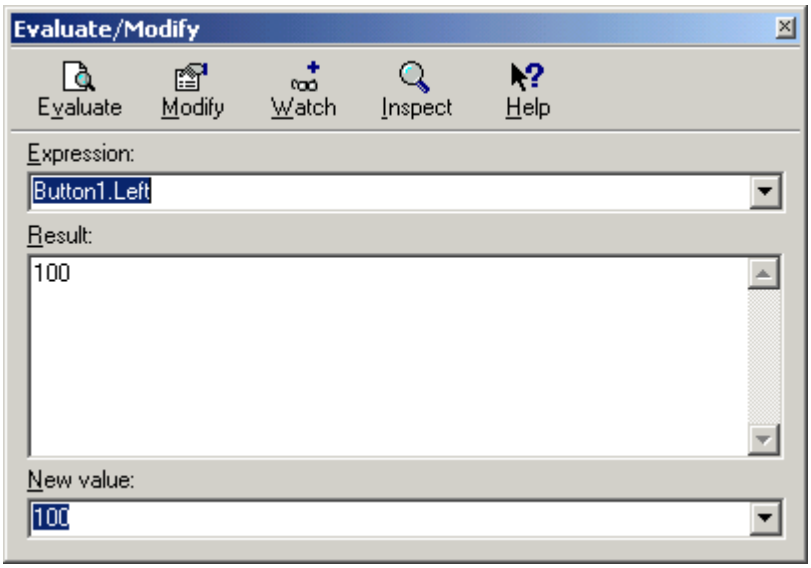


Figure 18.10: Using the Watch List window

You can set watches by using the Add Watch at Cursor command on the editor's local menu (or just press Ctrl+F5), but the faster technique in Delphi 5 is to drag a variable or expression from the source code editor to the Watch List window. When you add a watch, you'll need to choose the proper output format, and you may need to enter some text for more complex expressions. This is accomplished by double-clicking on the watch in the list, which opens the Watch Properties dialog box, or with the equivalent Edit Watch command of the shortcut menu.

> ☞Keep in mind that this window, like many other debugging windows, can be kept in view by docking it to the editor or by toggling its Stay on Top option.

The Local Variables Window

Another useful feature of Delphi is the Local Variables window. This window automatically displays the name and value of any local variables in the current procedure or function when you're paused at a breakpoint. For methods, you'll also see the implicit `Self` variable's private data. The Local Variables window is very similar to the Watch List window, but you don't have to set up its content, because it is automatically updated as you trace into a new function or method or stop on a breakpoint in a different one.

For any object references that appear in the Local Variables window (or the Watch List window) as well as displaying its detailed value on a single line, you can also open

Debug Inspector windows. To do so, either double-click on the variable in the Local Variables window or use the Inspect command of the shortcut menu in the Watch List window.

The Debug Inspector

Debug Inspector windows allow you to view data, methods, and properties of an object or component at run time, with a user interface similar to the design-time Object Inspector (as you can see in Figure 18.11). The main difference is that a Debug Inspector doesn't show just the published properties but the entire list of properties, methods, and local data fields of the object, including private ones. As already described, to activate a similar Inspector at debug time, you can select a visible identifier in the editor, activate the local menu, and select Debug ➢ Inspect.

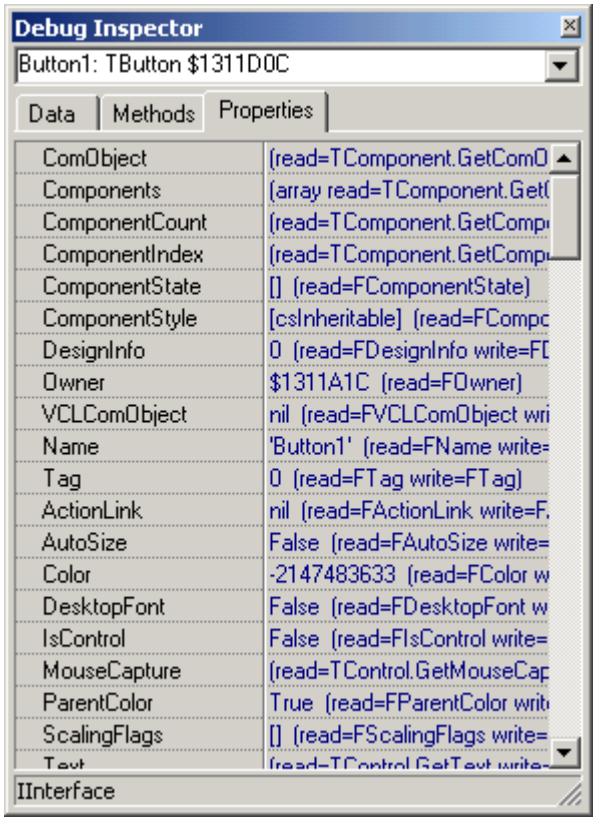


Figure 18.11: A Debug Inspector window showing the properties of a button

> ☞The Debug Inspector is similar to the Object Debugger component I've written for the *Delphi Developer's Handbook* and available on my Web site (www.marcocantu.com). This

component allows you to get a full list of the values of the published properties of a component at run time.

You'll see that the Debug Inspector shows the definition of the properties and not their values. To activate this, you have to select the property and click the small question mark button on the right. This computes the value, if available. You can also modify the object's data or property value.

If you are inspecting a `Sender` parameter, you can also cast the entire object to a different type, so that you'll be able to see its specific properties (without a cast you get information only about the generic `TObject` structure). When you are working on a component, you can drill down and inspect a sub-object, such as a font. You can use multiple Debug Inspector windows or use a single one and move back to the items you've inspected in the past. A drop-down list box at the top of the inspector maintains a history list of expressions for the current Debug Inspector.

Exploring Modules and Threads

Another important area of the debugger is related to exploring the overall structure of an application. The Modules window (see Figure 18.12) displays all the executable modules for the current application (usually the main executable file as well as the DLLs it uses). The right pane shows a list of the different procedure or function entry points of each module. The bottom pane displays a list of Pascal units that the module contains, if that information is known.

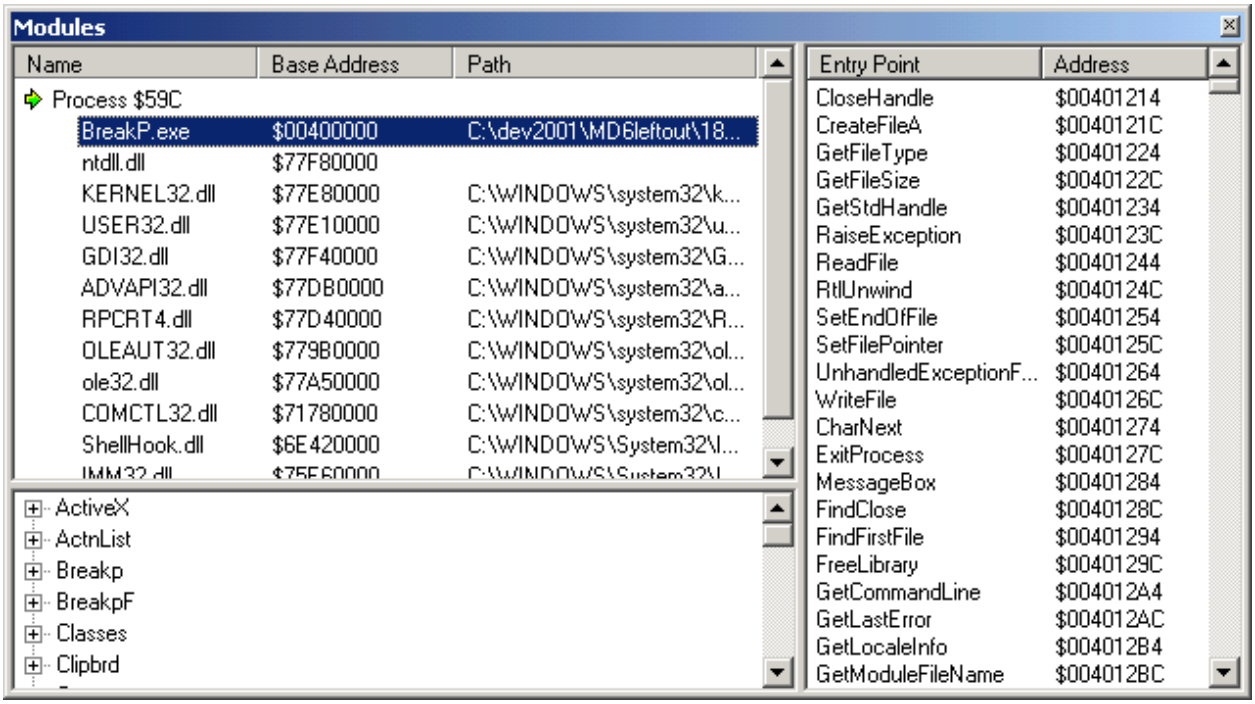


Figure 18.12: The Modules window (previous page...)

The Module window can be used to check the system DLLs and Delphi run-time packages required by a program, and it lets you explore how they relate in terms of exported and imported functions. While tools as TDump.exe or the Executable QuickView included in Windows can perform a static analysis of the EXE file to determine the DLLs it requires to run, the Modules window shows the current libraries in use, even if it's one you've dynamically loaded (see Chapter 14 for examples of dynamically loaded libraries). Remember that you can use the Module window to set a module breakpoint, one that will fire when the module is loaded by the system.

Another related window is the Thread Status window, which shows details about each thread of a multithreaded program. Figure 18.13 shows an example of this window. In this window, you can change the active thread as well as operate on the main process. Notice that this ability is particularly interesting when you are debugging two processes at the same time, a feature available only on the Windows NT platform.

[missing] -- Figure 18.13: The Thread Status window

The Event Log

Another handy debugger window, first introduced in Delphi 4, is the Event Log, which lets you monitor a number of system events: modules loading, breakpoints and their log messages, Windows messages, and custom messages sent by the application. Tracking program flow with a log can be invaluable. As an example, consider debugging an application where timing is important, so that it would be affected by stopping the program in a debugger. To avoid stopping the program, you can log debug information for later review.

To generate a direct log, you can embed calls to the `OutputDebugString` Windows API procedure in your program. This procedure accepts a string pointer and sends that string to a valid debug logging device. The Event Log window will capture and display the text you pass to the `OutputDebugString` procedure. Figure 18.14 shows an example of a debugging session made with the Event Log window. (By the way, calls to the `OutputDebugString` procedure appear in the log as text with the ODS prefix.) In the same figure you can also see the effect of some breakpoints logging the value of a variable, `J`. This output is taken from the OdsDemo example and was generated by the following code:

```
OutputDebugString (
  PChar ('Button2Click - I=' + IntToStr (I)));
```

> ☞Although the effects of calling `OutputDebugString` and logging a message as a result of a breakpoint might seem similar, there is a big difference. The breakpoints are external to the program (they're part of the debugger support); while the

direct strings must be added to the source code of the program, potentially changing it and introducing bugs. You might also want to remove this code for the final build, although you can use conditional compilation for this, as described later in the section "Using Conditional Compilation for Debug and Release Versions."

[missing] Figure 18.14: The output of calls to the `OutputDebugString` debugging procedure and the information logged by a breakpoint in the Event Log window

To capture the image in Figure 18.14, I've disabled the default breakpoint logging, which indicates when the program has stopped or restarted for a breakpoint. (As noted earlier, you can also specify logging as one of the breakpoint actions; disabling default breakpoint logging does not affect this type of logging. I've also disabled the process information (a new Delphi 5 option "Display Process Info with Event"), which is not particularly useful when debugging a single process.

You can configure the Event Log with this and other options in the Debugger Event Log Properties dialog box, shown in Figure 18.15. In addition to logging the text from calls to the `OutputDebugString` procedure, the breakpoint data, and the process information, the Event Log window can also capture all the Windows messages reaching the application. This provides an alternative to the WinSight program described later in this chapter.

[missing] Figure 18.15: You determine which events you wish to track using the Debugger Event Log Properties dialog box.

Down to the Metal: CPU and FPU views

There are two more debugger windows that are not for the faint-hearted. The CPU view and the new Delphi 5 FPU view show you what's going on inside the Central Processing Unit and the Floating Point Unit of the computer.

Using the CPU view during debugging lets you see a lot of system information: the values of the CPU registers, including special flags and a disassembly of the program (eventually with the Pascal source code included in comments). Similarly, the FPU view opens up more registers and status information regarding the floating point and MMX support of the newer Pentium class chips.

If you have a basic knowledge of assembly language, you can use this information to understand in detail how the Delphi compiler has translated your source code into the executable and see the effect of the Delphi optimizing compiler on the final code. Although it seems bare at first, the CPU window provides programmers with a lot of power. By right-clicking and using the shortcut menus, you can even change the value of CPU registers directly!

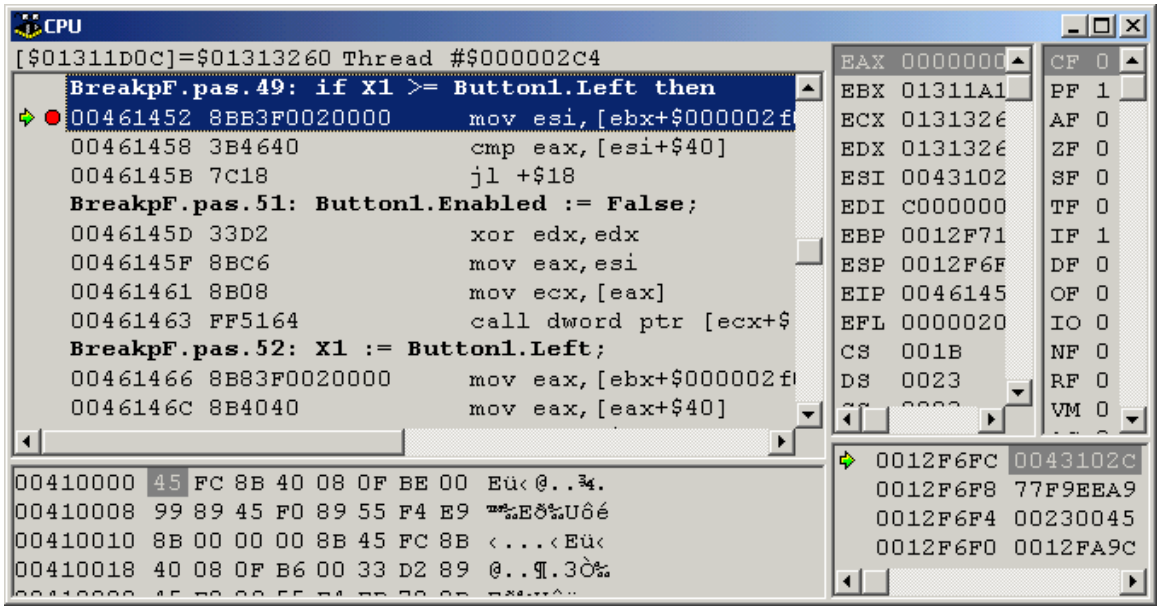


Figure 18.16: An example of the CPU view

Other Debugging Techniques

One common use of breakpoints is to find out when a program has reached a certain stage, but there are other ways to get this information. A common technique is to show simple messages (using the `ShowMessage` procedure) on specific lines, just for debugging purposes. There are many other manual techniques, such as changing the text of a label in a special form, writing to a log file, or adding a line in a list box or a memo field.

All of these alternatives serve one of two basic purposes: either they let you know that a certain statement of the code has been executed or they let you watch some values, in both cases without actually stopping the program. Conditional compilation, assertions, and message-flow spying are some of the techniques you can use to complement the features offered by the debugger.

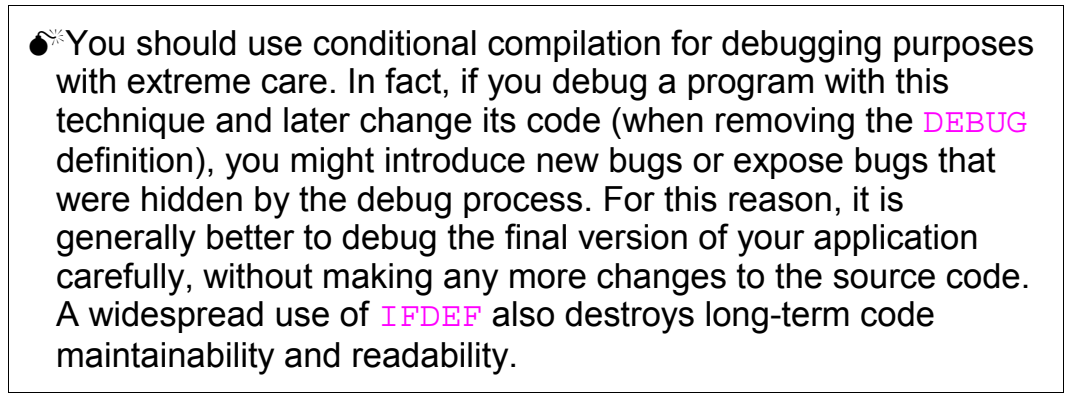
Using Conditional Compilation for Debug and Release Versions

Adding debugging code to an application is certainly interesting, as the OdsDemo example demonstrates, but this approach has a serious flaw. In the final version of the program, the one you give to your customers, you need to disable the debugging output, and you may need to remove all of the debugging code to reduce the size of the program and improve its speed. If you are a C/C++ programmer, however, you may have some ideas on how to remove program code automatically. The solution to this problem lies in a typical C technique known as *conditional compilation*. The idea is simple: you write some lines of code that you want to compile only in certain circumstances and skip on other occasions.

In Delphi, you can use some conditional compiler directives: `$IFDEF`, `$IFNDEF`, `$IFOPT`, `$ELSE`, and `$ENDIF`. For example, in the `Button2Click` event handler of the OdsDemo example, you'll find the following code:

```
{$IFDEF DEBUG}
  OutputDebugString (
    PChar ('Button2Click - I=' + IntToStr (I)));
{$ENDIF}
```

This code is included in the compilation only if there is a `DEBUG` symbol defined before the line or if the `DEBUG` symbol has been defined in the Project Options dialog box. Later on, you can remove the symbol definition, choose the Build All command from Delphi's Compile menu, and run it again. The size of the executable file will probably change slightly between the two versions because some source code is removed. Note that each time you change the symbol definitions in the Project Options dialog box, you need to rebuild the whole program. If you simply run it, the older version will be executed because the executable file seems up-to-date compared with the source files.



Using Assertions

Assertions are a technique you can use in Delphi for custom debugging. An assertion is basically an expression that should invariably be true, because it is part of the logic of the program. For example, I might assert that the number of users of my program should always be at least one because my program cannot run without any user. When an assertion is false, that means there is a flaw in the program code (in the *code,* not in the execution).

The only parameter of the `Assert` procedure is the Boolean condition you want to test (or *assert*). If the condition is met, the program can continue as usual; if the condition is not met (the assertion fails), the program raises an `EAssertionFailed` exception. Here is an example, from the Assert program:

```
procedure TForm1.BtnIncClick(Sender: TObject);
begin
  if Number < 100 then
    Inc (Number);
  ProgressBar1.Position := Number;
  // test the condition
  Assert ((Number > 0) and (Number <= 100));
end;
```

Another button on the Assert example's form generates code that is partially wrong, so that the assertion might actually fail, with the effect shown in Figure 18.17. Keep in mind that assertions are a debugging tool. They should be used to verify that the code of the program is correct. Users should never see assertions failing, no matter what happens in the program and which data they input, because if an assertion fails, there is probably an error in your code. To test for special error conditions, you should use exceptions, not assertions.

[missing] Figure 18.17: The error message of an assertion (from the Assert example)

Assertions are so closely linked to debugging and testing that you'll generally want to remove them using the `$ASSERTIONS` or `$C` compiler directive. Simply add the line `{$C-}` somewhere in a source code file, and that unit will be compiled without assertions. This doesn't just disable assertions, it actually removes the corresponding code from the program. You can also disable assertions from the Compiler page of the Project Options dialog box.

> 💣 Don't forget to rebuild your project after changing the assertions compiler settings, or the setting won't have effect.

Exploring the Message Flow

The integrated debugger provides common ways to explore the source code of a program. In Windows, however, this is often not enough. When you want to understand the details of the interaction between your program and the environment, you'll often need a tool to track the messages the system sends to your application. You can do this in the integrated debugger by using the Event Log and enabling Windows messages in the Event Log Properties dialog box. Note that this type of logging is not enabled by default.

The Event Log, however, is not very flexible, as you cannot choose the message categories or the destination windows: you get a full log of all of the messages for windows of your program, which is often a huge list. An alternative tool for tracing Windows messages is WinSight, a multipurpose tool included in Delphi. Other similar tools are available from many sources, including shareware, books, and magazine articles. I've built my own version, which you'll see shortly.

To become an expert Delphi programmer, you must learn to study the message flow following an input action by a user. As you know, Delphi programs (like Windows applications in general) are event-driven. Code is executed in response to an event. Windows messages are the key element behind Delphi events, although there isn't a one-to-one correspondence between the two. In Windows, there are many more messages than there are events in Delphi, but some Delphi events occur at a higher level than Windows messages. For example, Windows provides only a limited amount of support for mouse dragging, while Delphi components offer a full set of mouse-dragging events.

Using WinSight

WinSight is a Borland tool available in the Delphi/Bin directory. It can be used to build a hierarchical graph of existing windows and to display detailed information about the message flow. Of course, WinSight knows nothing about Delphi events, so you'll have to figure out the correspondence between many events and messages by yourself (or study the VCL source code, if you have it). WinSight can show you, in a readable format, all of the Windows messages that reach a window, indicating the destination window, its title or class, and its parameters. You can use the Options command from WinSight's Messages menu to filter out some of the messages and see only the groups you are interested in.

Usually, for Delphi programmers, spying the message flow can be useful when you are faced with some bugs related to the order of window activation and deactivation or to receiving and losing the input focus (`OnEnter` and `OnExit` events), particularly when message boxes or other modal windows are involved. This is quite a common problem area, and you can often see why things went wrong by looking at the message flow. You might also want to see the message flow when you are handling a Windows message

directly (instead of using event handlers). Using WinSight, you can get more information about when that message arrives and the parameters it carries.

A Look at Posted Messages

Another way to see the message flow is to trap some Windows messages directly in a Delphi application. If you limit this analysis to posted messages (delivered with `PostMessage`) and exclude sent messages (delivered with `SendMessage`), it becomes almost trivial, since we can use the `OnMessage` event of the `TApplication` class and the `TApplicationEvents` component. This event is intended to give the application a chance to filter the messages it receives and to handle certain messages in special ways. For example, you can use it to handle the messages for the window connected with the `Application` object itself, which has no specific event handlers, as we've done in the SysMenu2 example of Chapter 6.

In the MsgFlow example, however, we'll take a look at all of the messages extracted from the message queue of the application (that is, the posted messages). A description of each message is added to a list box covering the form of the example. For this list box, I've chosen Courier font because it is a nonproportional, or *monospaced*, font so that the output will be formatted with the fields aligned correctly in the list box. The speed buttons in the toolbar can be used to turn message viewing on and off, empty the list box, and skip consecutive, repeated messages. For example, if you move the mouse you get many consecutive `wm_MouseMove` messages, which can be skipped without losing much information.

To let you make some real tests, the program has a second form (launched by the fourth speed button), filled with various kinds of components (chosen at random). You can use this form to see the message flow of a standard Delphi window. Figure 18.18 shows an example of the output of the MsgFlow program when the second form is visible. The (lengthy) source code of the program is not described in the text, but it is fully available along with the examples of the book.

[missing] Figure 18.18: The MsgFlow program at run time, with a copy of the second form

Memory Problems

One of the biggest problems when debugging a Delphi program is to check what happens with the memory of the application and of the system. Two of the most common memory problems are *leaks* (not releasing unused memory, so that the program will use

much more memory than it actually needs) and memory overruns (using memory already in use or referencing an object that has already been deleted).

There are multiple approaches you can use to detect and solve these memory problems in Delphi, but there isn't much help you can receive from the integrated debugger. To detect these problems you can use the techniques described later in this section or the some of the third-party tools discussed at the end of this chapter. What I want to focus on is an overview of the different memory areas, so that you can better understand when these memory problems will surface and use a preventive approach to avoid them altogether.

Processes and Memory

It's not easy to review memory management for Delphi applications exhaustively, because there's so much information to consider. First, there's Windows memory management, which on Win32 platforms is fairly simple and robust for applications but a bit more complex for DLLs. On the application level, there's Delphi's own internal memory management.

In Win32 every application sees its local memory as a single large segment of 4GB, regardless of the amount of physical memory available. This is possible because the operating system maps the virtual memory addresses of each application into physical RAM addresses and swaps these blocks of memory to disk as necessary (automatically loading the proper page of the swap file in memory). This single, huge memory segment is managed by the operating system in chunks of 4KB each, called *pages*.

> 🕮Every process has its own private address space, totally separate from the others. This makes the operating system more robust than in the days of 16-bit Windows, when all applications shared a single address space. The drawback is that it is more difficult to pass data between applications.

In fact, in both 95/98 and NT, an application can directly manage only about half of its address space (2GB), while the other half is reserved for the operating system. Fortunately, 2GB is usually more than enough.

Another important element of Win32 memory management is virtual memory allocation. Besides allocating memory, a process can simply reserve memory for future use (using a low-level operation called virtual allocation). For example, in a Delphi application, you can use the `SetLength` procedure to reserve space for a string. Delphi does the same thing transparently when you create a huge array. This memory won't be allocated—just reserved for future use. In practice, this means that the memory subsystem won't use addresses in that range for other memory allocations.

Fortunately, most of the memory management, both at the application level and at the system level, is completely transparent to programmers. For this reason, you don't typically need to know the details of how memory pages work, and we won't explore that topic further here. Instead, we'll explore the status of a region of memory, something you might find very useful while writing and debugging an application.

Global Data, Stack, and Heap

The memory used by a specific Delphi application can be divided into two areas: code and data. Portions of the executable file of a program, of its resources (bitmaps and DFM files), and of the libraries used by the program are loaded in its memory space. These memory blocks are read-only, and they can be shared among multiple processes.

It is more interesting to look at the data portion. The data of a Delphi program is stored in three clearly distinct areas: the global memory, the stack, and the heap.

Global Memory

When the Delphi compiler generates the executable file, it determines the space required to store variables that exist for the entire lifetime of the program. Global variables declared in the interface or in the implementation portions of a unit fall into this category. Note that if a global variable is of a class type, only a 4-byte object reference is stored in the global memory.

You can determine the size of the global memory by using the Project ➢ Information menu item after compiling the program and looking at the value for data size. Figure 18.19, shows a usage of almost 6K of global data, which is not much considering it includes global data of the VCL and of your program.

[missing] Figure 18.19: The information about a compiled program shown by Delphi

Stack

The *stack* is a dynamic memory area, which is allocated and deallocated following the LIFO order: Last In, First Out. This means that the last memory object you've allocated will be the first to be deleted.

Stack memory is typically used by routines (procedure, function, and method calls) for passing parameters and their return values and for the variables you declare within a routine. Once a routine call is terminated, its memory area on the stack is released.

Remember, anyway, that using Delphi's default register-calling convention, the parameters are passed in CPU registers instead of the stack.

Windows application can reserve a large amount of memory for the stack. In Delphi you set this parameter in the linker page of the Project options. However, the default is generally OK. If you receive a stack full of error messages, this is probably because you have a function recursively calling itself forever, not because the stack space is too limited. The initial stack size is another piece of information provided by the Project ➢ Information menu item.

Heap

The *heap* is the area in which the allocation and deallocation of memory happens in random order. This means that if you allocate three blocks of memory in sequence, they can be destroyed later on in any order. The heap manager takes care of all the details, so you simply ask for new memory with `GetMem` or by calling a constructor to create an object, and Delphi will return a new memory block for you (possibly reusing memory blocks already discarded). Delphi uses the heap for allocating the memory of each and every object, the text of the strings, for dynamic arrays, and for other specific requests for dynamic memory.

Because it is dynamic, the heap is the memory area where programs generally have the most problems. Delphi uses numerous techniques to handle memory, including reference counting (for strings, dynamic arrays, or interface-type object variables) and ownership (for VCL components). Understanding these techniques and applying them properly is the foundation for a correct management of dynamic memory.

To check whether everything is working properly and to understand what is going wrong, Delphi's debugger is of little help. Both Windows and Delphi show the status of the memory (from two different perspectives), which lets you examine the current situation. Delphi also exposes its internal memory manager, so that you can hook into it or even replace it altogether. You can even change the memory handling for a specific class, by overriding its memory allocation and deallocation methods.

Tracking Memory

The Windows API includes a few functions that let us inspect the status of memory. The most powerful of these functions are part of the so-called ToolHelp API (nothing to do with Delphi's own ToolsAPI) and are platform-specific: They are available either on Windows 98 or on Windows NT—but not both.

If we want to remain on common ground we can use `GlobalMemoryStatus`, a function that allows us to inspect the status of the memory in the entire operating system. This function returns system information on the physical RAM, the page file (or swap

file), and the global address space. To demonstrate the use of this function, I've built the MemIcon program, which is described in Chapter 19 because its key element is to display the use of tray icons.

Generally, the information related to Windows memory status is of little interest to Delphi programmers, particularly if you compare it to the detailed information about Delphi's own memory manager returned by the `GetHeapStatus` VCL function. This function is defined by the System unit (or the ShareMem unit) and returns a structure with quite a bit of information: the address space that is virtually allocated; the space that's committed or uncommitted, physically allocated, free (distinguishing large and small memory blocks), and unused; and the total overhead of the memory manager.

You can see the data returned by this function in the Memory Status window, which the VclMem example displays as its About box. This window is visible in Figure 18.20. Its form hosts a string grid component, which is automatically updated when the program starts and by a timer (so you can keep this window open and see the information periodically updated). Aside from the memory information displayed, the program is not particularly interesting; you should add this form to one of your complex applications, so that you can test its memory status.

[missing] Figure 18.20: The Memory Status window of the simple VclMem example. You should add this window to your own programs to check how much memory they use over time.

> 📖This program is a reduced version of a test example from *Delphi Developer's Handbook* (Sybex, 1998). That book goes into more depth, discussing not only some cases in which the memory manager might cause problems but also how you can write a custom memory manager. You might do that to replace Delphi's memory management scheme with your own or to hook into the memory manager; for example, to count the number of memory blocks allocated and deallocated or checking for memory leaks.

Third-Party Tools

Delphi's integrated debugger, the stand-alone Turbo Debugger, and the remote debugger are great for helping you to trace source-code errors, but they won't help you much with memory problems, and they are still quite limited in some areas. Besides the techniques I've just discussed here, there are a few third-party tools that can be extremely helpful in tracing and solving memory problems and other debugging issues. In this

section, I'll just list a few of them and highlight their features in short, to give you an idea of what's available.

These notes are not intended as a full review, and I have no interest in endorsing any of these programs. I have simply noticed that their use can save you a lot of debugging time, and I would like to share this information.

Memory Sleuth

Memory Sleuth is produced by Turbo Power Software Company, Inc. (http://www.turbopower.com). It was originally developed for Delphi 1 by Per Larsen as MemMonD32. After compiling your Delphi program, you simply run it through Memory Sleuth (loading and running it from this environment instead of from the Delphi IDE).

The tool detects memory and Windows resource leaks, providing a detailed output with the source code lines causing problems. This tool hooks into the Delphi memory manager and monitors the allocation of Delphi objects but also checks the Windows memory status. Besides giving you a report about the problems, the program can also detect peak memory and resource usage, and even draw some nice graphs. A recent enhanced version adds profiling capabilities to the tool.

CodeSite

CodeSite is produced by Raize Software Solutions, Inc. (http://www.raize.com). The author is Ray Konopka (who also produced the Raize Components). In the words of its author, "CodeSite is an advanced Delphi debugging tool based on the time-honored approach of sending messages from an application to a message viewer. However, unlike all of its predecessors, CodeSite handles much more than simple strings."

In fact, you can send properties and entire objects to the debug window, which makes the entire operation very fast without being intrusive into the program code. Instead of using a standard debug window, you have to use the one provided by CodeSite, which stores extended information, and allows you, for example, to compare two snapshots of the same object done at different times.

BoundsChecker

BoundsChecker is a well-known Windows error detection tool from NuMega Technologies, Inc. (http://www.numega.com). The program has a long tradition with Microsoft C++ programmers and has been available for Delphi for a long time, as well. BoundsChecker monitors all Windows API calls made by your program or by the

VCL, tracking wrong parameters, resource leaks, stack and heap memory errors, and more. The tool validates the latest Windows APIs including Win32, ActiveX, DirectX, COM, Winsock, and Internet APIs.

You simply run the BoundsChecker program, load your executable file (which must be compiled with debug information and stack frames), and run it. Every time an error is detected, all the details are logged, so that you can trace the problems at the end of the debug session. You can also use BoundsChecker to test for compliance of your program with the different flavors of the Win32 API, if you think the problem is likely to have problems under Windows 98 or Windows NT.

What's Next?

In this chapter, you have seen that there are a number of tools you can use to debug a Delphi application, both by itself and in relation to the Windows system. Windows applications do not live in a world by themselves. They have a strong relationship with the system and, usually less directly, with the other applications that are running. The presence of other running Windows applications can affect the performance of your programs as well as their stability.

In the next chapter, I'll discuss a number of techniques related to printing, using resources, manipulating files, accessing the Clipboard, using Windows INI files, using the Registry, creating and linking help files, and creating an installation program, as well as new Delphi 5 features as TeamSource and the Integrated Translation Environment. Each technique represents a useful approach to solving a specific programming problem.