CHAPTER

19

Testing and Debugging

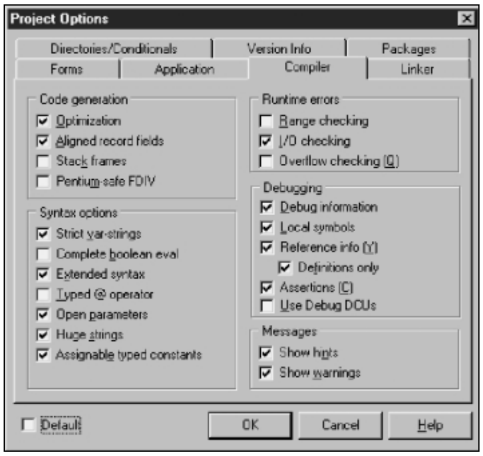
IN THIS CHAPTER

Some programmers in the industry believe that the knowledge and application of good programming practice make the need for debugging expertise unnecessary. In reality, however, the two complement each other, and whoever masters both will reap the greatest benefits. This is especially true when multiple programmers are working on different parts of the same program. It's simply impossible to completely remove the possibility of human error.

A surprising number of people say, "My code compiles all right, so I don't have any bugs, right?" Wrong. There's no correlation between whether a program compiles and whether it has bugs; there's a big difference between code that's syntactically correct and code that's logically correct and bug-free. Also, don't assume that because a particular piece of code worked yesterday or on another system that it's bug-free. When it comes to hunting software bugs, everything should be presumed guilty until proven innocent.

During the development of any application, you should allow the compiler to help you as much as possible. You can do this in Delphi by enabling all the runtime error-checking options in Project, Options, Compiler, as shown in Figure 19.1, or by enabling the necessary directives in your code. Additionally, you should have the Show Hints and Show Warnings options enabled in that same dialog box in order to receive more information on your code. It's common for a developer to spend needless hours trying to track down "that impossible bug," when he or she could have found the error immediately by simply employing these effective compiler-aided tools. (Of course, the authors would never be guilty of failing to remember to use these aids. You believe us, right?)

Table 19.1 describes the different runtime error options available through Delphi.



Code generation
- ✓ Optimization
- ✓ Aligned record fields
- ☐ Stack frames
- ☐ Pentium-safe FDIV

Syntax options
- ✓ Strict var-strings
- ☐ Complete boolean eval
- ✓ Extended syntax
- ☐ Typed @ operator
- ✓ Open parameters
- ✓ Huge strings
- ✓ Assignable typed constants

Runtime errors
- ☐ Range checking
- ✓ I/O checking
- ☐ Overflow checking (Q)

Debugging
- ✓ Debug information
- ✓ Local symbols
- ✓ Reference info (Y)
  - ✓ Definitions only
- ✓ Assertions (C)
- ☐ Use Debug DCUs

Messages
- ✓ Show hints
- ✓ Show warnings

☐ Default

FIGURE 19.1
The Compiler page of the Project Options dialog box.

TABLE 19.1 Delphi Runtime Errors

Runtime Error	*Directive*	*Function*
Range Checking	`{$R+}`	Checks to ensure that you don't index an array or string beyond its bounds and that assignments don't assign a value to a scalar variable that's outside its range.
I/O Checking	`{$I+}`	Checks for an input/output error after every I/O call (`ReadLn()` and `WriteLn()`, for example). This almost always should be enabled.
Overflow Checking	`{$Q+}`	Checks to ensure that calculation results are not larger than the register size.

TIP

Keep in mind that each of these runtime errors exacts a performance penalty on your application. Therefore, once you're out of the debugging phase of development and are ready to ship a final product, you can improve performance by disabling some of the runtime error checks. It's common practice for developers to disable all of them except I/O Checking for the final product.

Common Program Bugs

This section shows some commonly made mistakes that cause programs to fail or crash. If you know what to look for when you're debugging code, you can lessen the time needed to find errors.

Using a Class Variable Before It's Created

One of the most common bugs that creeps up when you develop in Delphi occurs because you've used a class variable before it has been created. For example, take a look at the following code:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  MyStringList: TStringList;
begin
  MyStringList.Assign(ListBox1.Items);
end;
```

The `TStringList` class `MyStringList` has been declared; however, it's used before it's instantiated. This is a sure way to cause an access violation. You must be sure to instantiate any class variables before you try to use them. The following code shows the correct way to instantiate and use a class variable. However, it also introduces another bug. Can you see it?

```
procedure TForm1.Button1Click(Sender: TObject);
var
  MyStringList: TStringList;
begin
  MyStringList := TStringList.Create;
  MyStringList.Assign(ListBox1.Items);
end;
```

If your answer was, “You didn’t free your `TStringList` class,” you’re correct. This won’t cause your program to fail or crash, but it will eat up memory because, every time you call this method, another `TStringList` is created and thrown away, thereby leaking memory. Although the Win32 API will free all memory allocated by your process at the time it terminates, leaking memory while running an application can cause serious problems. For example, a leaky application will continue to eat more and more of the system’s memory resources as it runs, causing the OS to have to perform more disk swapping, which ultimately slows down the entire system.

The corrected version of the preceding code listing is shown in the following code (minus a necessary enhancement discussed in the next topic):

```
procedure TForm1.Button1Click(Sender: TObject);
var
  MyStringList: TStringList;
begin
  MyStringList := TStringList.Create;               // Create it!
  MyStringList.Assign(ListBox1.Items);              // Use it!
  { Do your stuff with your TStringList instance }
  MyStringList.Free;                                // Free it!
end;
```

Ensuring That Class Instances Are Freed

Suppose that in the previous code example, an exception occurs just after `TStringList` is created. The exception would cause the flow of execution to immediately exit the procedure, and none of the procedure’s remaining code would be executed, which would cause a memory loss. Make sure your class instances are freed, even if an exception occurs, by using a `try..finally` construct, as shown here:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  MyStringList: TStringList;
begin
  MyStringList := TStringList.Create;                 // Create it!
  try
    MyStringList.Assign(ListBox1.Items);              // Use it!
    { Do your stuff with your TStringList instance }
  finally
    MyStringList.Free;                                // Free it!
  end;
end;
```

After you read the section “Breakpoints” later in the chapter, try an experiment and place the following line right after the line where you assign the `ListBox1` items to the `TStringList`:

```
raise Exception.Create('Test Exception');
```

Then place a breakpoint at the beginning of the method’s code and step through the code. You’ll see that `TStringList` still gets freed, even after the exception is raised.

Taming the Wild Pointer

The *wild pointer bug* is a common error that clobbers some part of memory when you use the pointer to write to memory. The wild pointer genus has two common species: the uninitialized pointer and the stale pointer.

An *uninitialized pointer* is a pointer variable that’s used before memory has been allocated for it. When such a pointer is used, you end up writing to whatever address happens to live at the location of the pointer variable. The following code example illustrates an uninitialized pointer:

```
var
  P: ^Integer;
begin
  P^ := 1971;  // Eeek! P is uninitialized!
```

A *stale pointer* is a pointer that references an area of memory that was once properly allocated but has been freed. The following code shows a stale pointer:

```
var
  P: ^Integer;
begin
  New(P);
  P^ := 1971;
  Dispose(P);
  P^ := 4;  // Eeek! P is stale!
```

If you’re lucky, you’ll receive an access violation when you attempt to write to a wild pointer. If you’re not so lucky, you’ll end up writing over data used by some other part of your application. This type of error is absolutely no fun to debug. On one machine, the pointer may appear to run just fine until you transfer it to another machine (and maybe make a few code changes in the process), where it begins to malfunction. This may lead you to believe that the recent changes you made are faulty or that the second machine has a hardware problem. Once you’ve fallen into this trap, all the good programming practice in the world won’t save you. You may start adding instances of `ShowMessage()` to portions of your code in an attempt to find the problem, but this serves only to modify the code's location in memory and might cause the bug to move around—or worse, disappear! Your best defense against wild pointer bugs is to avoid them in the first place. Whenever you need to work with pointers and manual memory allocation, make sure you check and double-check your algorithms to avoid the silly mistake that may introduce a bug.

Using Uninitialized PChar-Type Variables

You'll often see wild pointer errors when you use `PChar`-type variables. Because a `PChar` is just a pointer to a string, you have to remember to allocate memory for the `PChar` by using the `StrAlloc()`, `GetMem()`, `StrNew()`, `GlobalAlloc()`, or `VirtualAlloc()` function, as well as using the `FreeMem()`, `StrDispose()`, `GlobalFree()`, or `VirtualFree()` function to free it.

TIP

You can avoid potential bugs in your program by using `string`-type variables where possible, instead of `PChar`s. You can typecast a `string` to a `PChar`, so the code involved is simple, and because `string`s are automatically allocated and freed, you don't have to concern yourself with memory allocation.

This holds true especially for Delphi 1.0 applications that you're porting to 32-bit Delphi. In Delphi 1.0, `PChar`s are a necessary evil. In 32-bit Delphi, they're necessary only on rare occasions. Take the time to move to `string`s as you port your applications to 32-bit Delphi.

Dereferencing a nil Pointer

In addition to the wild pointer, another common mistake is dereferencing a `nil` (zero-value) pointer. Dereferencing a `nil` pointer always causes the operating system to issue an access violation error. Although this isn't an error that you want to have in your application, it's generally not fatal. Because it doesn't actually corrupt memory, it's safe to use exception handling to take care of the exception and move along. The sample procedure in the following code listing illustrates this point:

```
procedure I_AV;
var
  P: PByte;
begin
  P := Nil;
  try
    P^ := 1;
  except
```

```
    on EAccessViolation do
      MessageDlg('You can''t do that!!', mtError, [mbOk], 0);
  end;
end;
```

If you put this procedure in a program, you'll see that the message dialog box appears to inform you of the problem, but your program continues to run.

Using the Integrated Debugger

Delphi provides a feature-rich debugger built right into the IDE. Most of the facilities of the integrated debugger can be found on the Run menu. These facilities include all the features you would expect of a professional debugger, including the ability to specify command-line parameters for your application, set breakpoints, perform trace and step, add and view watches, evaluate and modify data, and view call stack information.

Using Command-Line Parameters

If your program is designed to use command-line parameters, you can specify them in the Run Parameters dialog box. In this dialog box, simply type the parameters as you would on the command line or in the Windows Start menu's Run dialog box.

Breakpoints

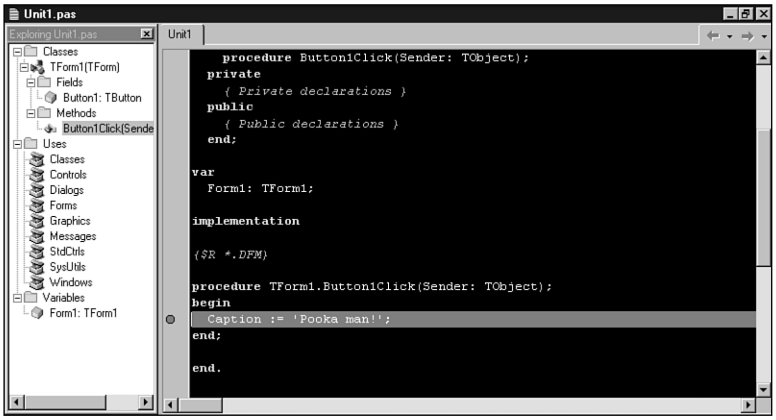
Breakpoints enable you to suspend the execution of your program whenever a certain condition is met. The most common type of breakpoint is a *source breakpoint*, which occurs when a particular line of code is about to be executed. You can set a source breakpoint by clicking to the far left of a line of code in the Code Editor, by using the local menu, or by selecting Run, Add Breakpoint. Whenever you want to see how your program is behaving inside a particular procedure or function, just set a breakpoint on the first line of code in that routine. Figure 19.2 shows a source breakpoint set on a line of program code.

Conditional Breakpoints

You can add additional information to a source breakpoint to suspend the execution of your program when some condition occurs in addition to when a line of code is reached. A typical example is when you want to examine the code inside a loop construct. You probably don't want to suspend and resume execution every time your code passes through the loop, especially if the loop occurs hundreds, or perhaps thousands, of times. Instead of continually pressing the F9 key to run, just set a breakpoint to occur whenever a variable reaches a certain value. For example, in a new project, place a `TButton` on the main form and add the following code to the button's event handler:

```
procedure TForm1.Button1Click(Sender: TObject);
var
```

```
  I: Integer;
begin
  for I := 1 to 100 do
  begin
    Caption := IntToStr(I);          // update form
    Button1.Caption := IntToStr(I); // update button
    Application.ProcessMessages;    // let updates happen
  end;
end;
```



```
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
begin
  Caption := 'Pooka man!';
end;

end.
```

FIGURE 19.2
A source breakpoint set in the Code Editor.

Now set a breakpoint on the following line:

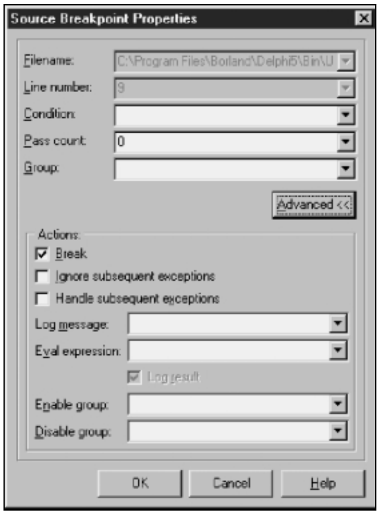
```
Caption := IntToStr(I);          // update form
```

After you've set a breakpoint, select View, Debug Windows, Breakpoints, which will bring up a Breakpoint List dialog box. Your breakpoint should show up in this list. Right-click your breakpoint and select Properties from the local menu. This will invoke the Edit Breakpoint dialog box, as shown in Figure 19.3. In the Condition input line, enter `I = 50` and select OK. This will cause the breakpoint that you previously set to suspend program execution only when the variable `I` contains the value `50`.

TIP

Figure 19.3 provides a glimpse into the breakpoint actions feature, which is new to Delphi 5. Breakpoint actions enable you to specify the exact behavior of the debugger when a breakpoint is encountered. These actions are controlled using the three checkboxes shown in the figure. Break, as you might imagine, instructs the debugger to break when the breakpoint is encountered. Ignore Subsequent Exceptions causes the debugger to refrain from breaking when exceptions are encountered from the breakpoint forward. Handle Subsequent Exceptions causes the debugger to resume the default behavior of breaking when exceptions are encountered from the breakpoint forward.

The latter two options are designed to be used in tandem. If you have a particular bit of code that is causing you problems by raising exceptions in the debugger and you don't want to be notified about it, you can use these breakpoint options to instruct the debugger to ignore exceptions before entering the code block and begin handling exceptions once again after leaving the block.


Source Breakpoint Properties

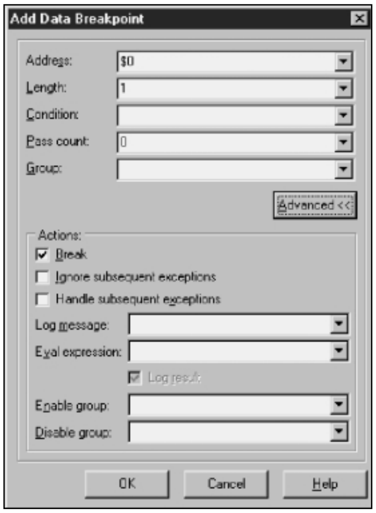
Filename: C:\Program Files\Borland\Delphi5\Bin\U
Line number: 9
Condition:
Pass count: 0
Group:

Actions:
[✓] Break
[ ] Ignore subsequent exceptions
[ ] Handle subsequent exceptions
Log message:
Eval expression:
[✓] Log result
Enable group:
Disable group:

FIGURE 19.3
The Edit Breakpoint dialog box.

Data Breakpoints

Data breakpoints are breakpoints you can set to occur when memory at a particular address is modified. This is useful for low-level debugging, when you need to track down bugs that perhaps occur when a variable gets assigned. You can set data breakpoints by selecting Run, Add Breakpoint, Data Breakpoint from the main menu or by using the local menu on the Breakpoint List dialog box. This invokes the Add Data Breakpoint dialog box, as shown in Figure 19.4. In this dialog box, you can enter the start address of the area of memory you want to monitor and the length (number of bytes) to monitor after that address. By specifying the number of bytes, you can watch anything from a `Char` (one byte) to an `Integer` (four bytes) to an array or record (any number of bytes). In a manner similar to source breakpoints, the Add Data Breakpoint dialog box also allows you to enter an expression that will be evaluated when the memory region is written to so that you can find those bugs that occur on the *n*th time a memory region is set. If you want the debugger to break when a specific variable is modified, just enter the name of the variable in the address field.



Address: $0
Length: 1
Condition:
Pass count: 0
Group:
[✓] Break
[ ] Ignore subsequent exceptions
[ ] Handle subsequent exceptions
Log message:
Eval expression:
[✓] Log result
Enable group:
Disable group:

FIGURE 19.4
The Add Data Breakpoint dialog box.

Address Breakpoints

An *address breakpoint* is a breakpoint you can set to occur when code residing at a particular address is executed. These types of breakpoints are normally set from the local menu in the CPU view when you can't set a source breakpoint because you don't have the source code for a particular module. As with other types of breakpoints, you can also specify a condition for address breakpoints in order to fine-tune your breakpoints.

Module Load Breakpoints

As you can probably surmise from the name, *module load breakpoints* enable you to set breakpoints that occur when a specified module is loaded in the debugged application's process. This allows you to be notified immediately when a DLL or package is loaded by an application. The most common place to set module load breakpoints is the local menu in the Modules window, but they can also be set by using the Run, Add Breakpoint item on the main menu.

Breakpoint Groups

Breakpoint groups are one of the most powerful and time-saving features the integrated debugger offers. Using groups, any breakpoint can be set up to enable or disable any other breakpoint so that a very complex algorithm of breakpoints can be created to find very specific bugs. Suppose you suspect that a bug shows up in your `Paint()` method only after you choose a particular menu option. You could add a breakpoint to the `Paint()` method, run the program, and constantly tell the debugger to continue when you get barraged with hundreds of calls to your `Paint()` method. Alternatively, you could keep that breakpoint on your `Paint()` method, disable it so that it doesn't fire, and then add another breakpoint to your menu-select event handler to enable the `Paint()` method breakpoint. Now you can run full speed in the debugger and not break in your `Paint()` handler until after you select the menu choice.

Executing Code Line by Line

You can execute code line by line by using either the Step Over or Trace Into option (F8 and F7 keys, respectively, in the Default and IDE classic keymapping). Trace Into steps into your procedures and functions as they're called; Step Over executes the procedure or function immediately without stepping into it. Typically, you use these options after stopping somewhere in your code with a breakpoint. Get to know the F7 and F8 keys; they are your friends.

You can also tell Delphi to run your program up to the line that the cursor currently inhabits by using the Run To Cursor (F4) option. This is particularly useful when you want to bypass a loop that's iterated many times, in which case using F7 or F8 becomes tedious. Keep in mind that you can set breakpoints at any time in the Code Editor—even as your program executes; you don't have to set all the breakpoints up front.

TIP

If you accidentally step into a function that will be very difficult or time-consuming to step out of, choose Run, Run Until Return from the main menu to cause the debugger to break after the current procedure or function returns.

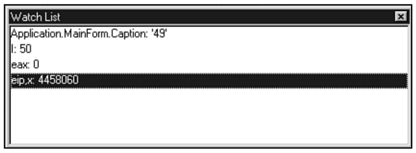
You can breakpoint your code dynamically by using the Program Pause option. This option often helps you determine whether your program is in an infinite loop. Keep in mind that VCL code is being run most of your program's life, so you often won't stop on a line of your program's code with this option.

TIP

When you debug your application, you've probably noticed the blue dots shown in the "gutter" on the left side of the Code Editor window. One of these blue dots is shown next to each line of code for which machine code is generated. You can't set a breakpoint on or step to a particular line of code if it doesn't have a blue dot next to it because no machine code is associated with the line.

Using the Watch Window

You can use the Watch window to track the values of your program's variables as your code executes. Keep in mind that you must be in a code view of your program (a breakpoint should be executed) for the contents of the Watch window to be accurate. You can enter an Object Pascal expression or register name into the Watch window. This is shown in Figure 19.5.



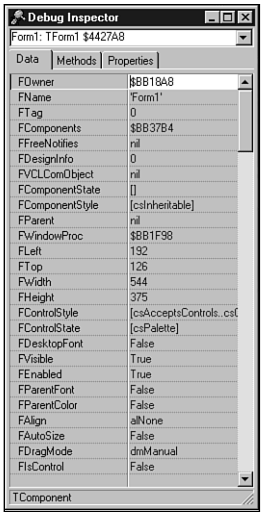
```
Application.MainForm.Caption: '49'
I: 50
eax: 0
eip,x: 4458060
```

FIGURE 19.5
Using the Watch List window.

Debug Inspectors

A debug inspector is a kind of data inspector that's perhaps easier to use and more powerful in some ways than the Watch window. To use this feature, select Run, Inspect while debugging an application. This will invoke a simple dialog box into which you can enter an expression. Click OK, and you'll be presented with a Debug Inspector window for the expression you entered. For example, Figure 19.6 shows a Debug Inspector for the main form of a do-nothing Delphi application.

The Debug Inspector window provides a means for conveniently viewing data that consists of many individual elements, such as classes and records. Click the ellipses on the right of the value column in the Inspector to modify the value of a field. You can even drill down into record or class data members by double-clicking a field of this type in the list.



Form1: TForm1 $4427A8

FOwner	$BB18A8
FName	'Form1'
FTag	0
FComponents	$BB37B4
FFreeNotifies	nil
FDesignInfo	0
FVCLComObject	nil
FComponentState	[]
FComponentStyle	[csInheritable]
FParent	nil
FWindowProc	$BB1F98
FLeft	192
FTop	126
FWidth	544
FHeight	375
FControlStyle	[csAcceptsControls..cs(
FControlState	[csPalette]
FDesktopFont	False
FVisible	True
FEnabled	True
FParentFont	False
FParentColor	False
FAlign	alNone
FAutoSize	False
FDragMode	dmManual
FIsControl	False

TComponent

FIGURE 19.6
Inspecting a form using a Debug Inspector.

Using the Evaluate and Modify Options

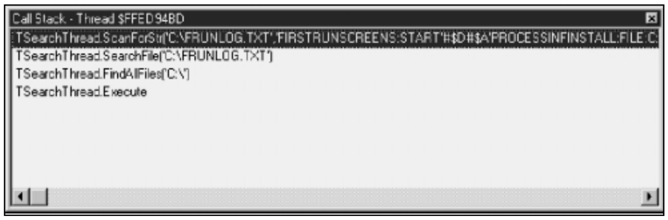
The Evaluate and Modify options enable you to inspect and change the contents of variables, including arrays and records, on the fly as your application executes in the integrated debugger. Keep in mind that this feature doesn't enable you to access functions or variables that are out of scope.

CAUTION

Evaluating and modifying variables is perhaps one of the more powerful features of the integrated debugger, but with that power comes the responsibility of having direct access to memory. You must be careful when changing the values of variables, because changes can affect the behavior of your program later.

Accessing the Call Stack

You can access the call stack by choosing View, Debug Windows, Call Stack. This enables you to view function and procedure calls along with the parameters passed to them. The call stack is useful for seeing a road map of functions that were called up to the current point in your source code. Figure 19.7 shows a typical view of the Call Stack window.



```
Call Stack - Thread $FFED94BD
TSearchThread.ScanForStr('C:\FRUNLOG.TXT','FIRSTRUNSCREENS:START'#$D#$A'PROCESSINFINSTALL:FILE:C
TSearchThread.SearchFile('C:\FRUNLOG.TXT')
TSearchThread.FindAllFiles('C:\')
TSearchThread.Execute
```

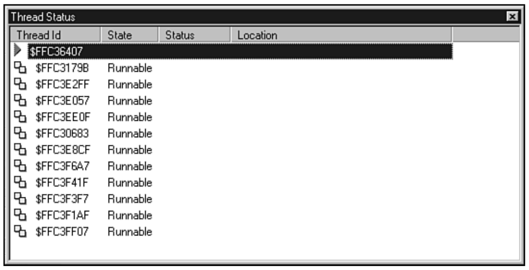
FIGURE 19.7
The Call Stack window.

TIP

To view any procedure or function listed in the Call Stack window, simply right-click inside the window. This is a good trick for getting back to a function when you accidentally trace in too far.

Viewing Threads

If your application makes use of multiple threads, the integrated debugger allows you to obtain information on the various threads in your application through the Thread Status window. Select View, Debug Windows, Threads from the main menu to invoke this window. When your application is paused (has hit a breakpoint), you can use the local menu provided by this window to make another thread current or to view the source associated with a particular thread. Remember that whenever you modify the current thread, the next run or step command you issue is relative to that thread. Figure 19.8 shows the Thread Status window.



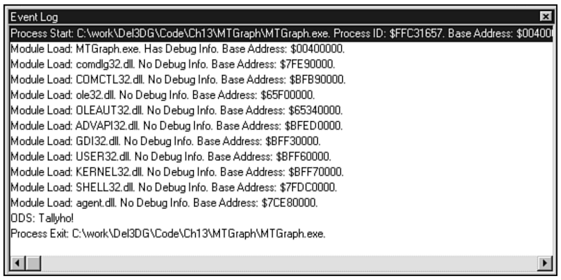
Thread Status

Thread Id	State	Status	Location
$FFC36407			
$FFC3179B	Runnable		
$FFC3E2FF	Runnable		
$FFC3E057	Runnable		
$FFC3EE0F	Runnable		
$FFC30683	Runnable		
$FFC3E8CF	Runnable		
$FFC3F6A7	Runnable		
$FFC3F41F	Runnable		
$FFC3F3F7	Runnable		
$FFC3F1AF	Runnable		
$FFC3FF07	Runnable		

FIGURE 19.8
The Thread Status window.

Event Log

The Event Log provides a place into which the debugger will log a record for the occurrence of various events. The Event Log, shown in Figure 19.9, is accessible from the View, Debug menu. You can configure the Event Log by using its local menu or the Debugger page of the Tools, Environment Options dialog box.



```
Process Start: C:\work\Del3DG\Code\Ch13\MTGraph\MTGraph.exe. Process ID: $FFC31657. Base Address: $00400
Module Load: MTGraph.exe. Has Debug Info. Base Address: $00400000.
Module Load: comdlg32.dll. No Debug Info. Base Address: $7FE90000.
Module Load: COMCTL32.dll. No Debug Info. Base Address: $BFB90000.
Module Load: ole32.dll. No Debug Info. Base Address: $65F00000.
Module Load: OLEAUT32.dll. No Debug Info. Base Address: $65340000.
Module Load: ADVAPI32.dll. No Debug Info. Base Address: $BFED0000.
Module Load: GDI32.dll. No Debug Info. Base Address: $BFF30000.
Module Load: USER32.dll. No Debug Info. Base Address: $BFF60000.
Module Load: KERNEL32.dll. No Debug Info. Base Address: $BFF70000.
Module Load: SHELL32.dll. No Debug Info. Base Address: $7FDC0000.
Module Load: agent.dll. No Debug Info. Base Address: $7CE80000.
ODS: Tallyho!
Process Exit: C:\work\Del3DG\Code\Ch13\MTGraph\MTGraph.exe.
```

FIGURE 19.9
The Event Log.

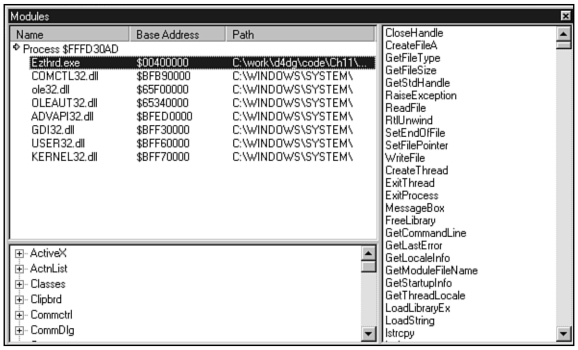
The types of events you can log include process information such as process start, process stop, and module load debugger breakpoints, as well as Windows messages sent to the application and application output using `OutputDebugString()`.

TIP

The `OutputDebugString()` API function provides a handy means to help you debug applications. The single parameter to `OutputDebugString()` is a `PChar`. The string passed in this parameter will be passed on to the debugger, and in the case of Delphi, the string will be added to the Event Log. This allows you to keep track of variable values or similar debug information without having to use watches or displaying intrusive debug dialog boxes.

Modules View

The Modules view enables you to obtain information on all the modules (`EXE`, `DLL`, `BPL`, and so on) loaded into the debugged application's process. Shown in Figure 19.10, this window provides you with a list of who's who in your application's process, permits you to set module load breakpoints, and provides you with various types of information on each module.

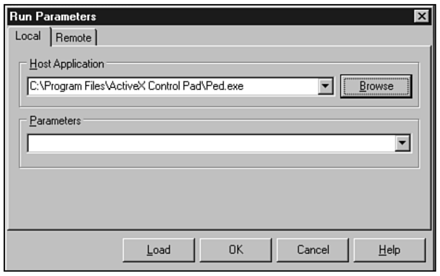


```
Modules
Name              Base Address   Path
Process $FFFD30AD
  Ezthrd.exe      $00400000      C:\work\d4dg\code\Ch11\...
  COMCTL32.dll    $BFB90000      C:\WINDOWS\SYSTEM\
  ole32.dll       $65F00000      C:\WINDOWS\SYSTEM\
  OLEAUT32.dll    $65340000      C:\WINDOWS\SYSTEM\
  ADVAPI32.dll    $BFED0000      C:\WINDOWS\SYSTEM\
  GDI32.dll       $BFF30000      C:\WINDOWS\SYSTEM\
  USER32.dll      $BFF60000      C:\WINDOWS\SYSTEM\
  KERNEL32.dll    $BFF70000      C:\WINDOWS\SYSTEM\

ActiveX
ActnList
Classes
Clipbrd
Commctrl
CommDlg

CloseHandle
CreateFileA
GetFileType
GetFileSize
GetStdHandle
RaiseException
ReadFile
RtlUnwind
SetEndOfFile
SetFilePointer
WriteFile
CreateThread
ExitThread
ExitProcess
MessageBox
FreeLibrary
GetCommandLine
GetLastError
GetLocaleInfo
GetModuleFileName
GetStartupInfo
GetThreadLocale
LoadLibraryEx
LoadString
lstrcpy
```

FIGURE 19.10
The Modules view.

DLL Debugging

The Delphi integrated debugger provides you with the ability to debug your DLL projects using any arbitrary application as the host. If fact, it's quite easy. Open your DLL project and select Run, Parameters from the main menu. Then specify a host application in the Run Parameters dialog box, as shown in Figure 19.11.



Run Parameters

Local | Remote

Host Application: C:\Program Files\ActiveX Control Pad\Ped.exe

Parameters:

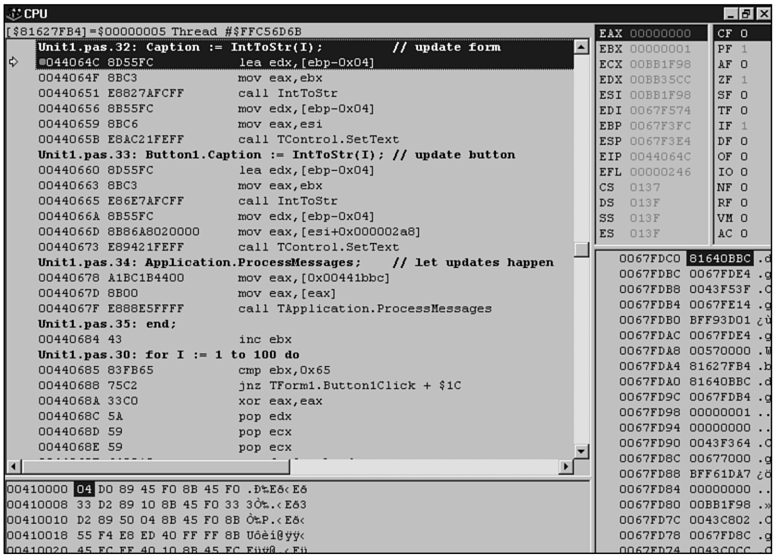
FIGURE 19.11
Specifying a host application.

The host application is an executable file that loads and uses the DLL you're currently debugging. After specifying a proper host application, you can use the integrated debugger much as you would for debugging a normal executable; you can set breakpoints, step, trace, and so on.

This feature is most useful for debugging ActiveX controls and in-process COM servers that are executed from within the context of another process. For example, you can use this feature to debug your ActiveX control from within Visual Basic.

The CPU View

The CPU view, found by selecting View, Debug Windows, CPU from the main menu, provides a developer's-eye view of what's going on inside the machine's CPU. The CPU view consists of five panes of information: the CPU pane, the Memory Dump pane, the Register pane, the Flags pane, and the Stack pane (see Figure 19.12). Each of these panes enables the user to view important aspects of the processor as an aid to debugging.



```
[$81627FB4]=$00000005 Thread #$FFC56D6B
Unit1.pas.32: Caption := IntToStr(I);         // update form
0044064C 8D55FC            lea edx,[ebp-0x04]
0044064F 8BC3              mov eax,ebx
00440651 E8827AFCFF        call IntToStr
00440656 8B55FC            mov edx,[ebp-0x04]
00440659 8BC6              mov eax,esi
0044065B E8AC21FEFF        call TControl.SetText
Unit1.pas.33: Button1.Caption := IntToStr(I); // update button
00440660 8D55FC            lea edx,[ebp-0x04]
00440663 8BC3              mov eax,ebx
00440665 E86E7AFCFF        call IntToStr
0044066A 8B55FC            mov edx,[ebp-0x04]
0044066D 8B86A8020000      mov eax,[esi+0x000002a8]
00440673 E89421FEFF        call TControl.SetText
Unit1.pas.34: Application.ProcessMessages;    // let updates happen
00440678 A1BC1B4400        mov eax,[0x00441bbc]
0044067D 8B00              mov eax,[eax]
0044067F E888E5FFFF        call TApplication.ProcessMessages
Unit1.pas.35: end;
00440684 43                inc ebx
Unit1.pas.30: for I := 1 to 100 do
00440685 83FB65            cmp ebx,0x65
00440688 75C2              jnz TForm1.Button1Click + $1C
0044068A 33C0              xor eax,eax
0044068C 5A                pop edx
0044068D 59                pop ecx
0044068E 59                pop ecx
```

```
EAX 00000000   CF 0
EBX 00000001   PF 1
ECX 00BB1F98   AF 0
EDX 00BB35CC   ZF 1
ESI 00BB1F98   SF 0
EDI 0067F574   TF 0
EBP 0067F3FC   IF 1
ESP 0067F3E4   DF 0
EIP 0044064C   OF 0
EFL 00000246   IO 0
CS  0137       NF 0
DS  013F       RF 0
SS  013F       VM 0
ES  013F       AC 0
```

```
00410000 04 D0 89 45 F0 8B 45 F0 .Ð‰Eð‹Eð
00410008 33 D2 89 10 8B 45 F0 33 3Ò‰.‹Eð3
00410010 D2 89 50 04 8B 45 F0 8B Ò‰P.‹Eð‹
00410018 55 F4 E8 ED 40 FF FF 8B Uôèí@ÿÿ‹
00410020 45 FC FF 40 10 8B 45 FC Eüÿ@.‹Eü
```

```
0067FDC0 81640BBC .d
0067FDBC 0067FDE4 .g
0067FDB8 0043F53F .C
0067FDB4 0067FE14 .g
0067FDB0 BFF93D01 ¿ù
0067FDAC 0067FDE4 .g
0067FDA8 00570000 .W
0067FDA4 81627FB4 .b
0067FDA0 81640BBC .d
0067FD9C 0067FDB4 .g
0067FD98 00000001 ..
0067FD94 00000000 ..
0067FD90 0043F364 .C
0067FD8C 00677000 .g
0067FD88 BFF61DA7 ¿ö
0067FD84 00000000 ..
0067FD80 00BB1F98 .»
0067FD7C 0043C802 .C
0067FD78 0067FD8C .g
0067FD74 0043C0CC .C
```

FIGURE 19.12
The CPU view.

The CPU pane shows the opcodes and mnemonics of the disassembled assembly code that's being executed. You can position the CPU pane at any address in your process to view instructions, or you can set the current instruction pointer to any new location, from which execution then continues. It's helpful to be able to understand the assembly code that the CPU pane displays, and experienced developers will attest to the fact that many bugs have been found and exterminated by examining the assembly code generated for a routine and realizing that it wasn't performing the desired operation. Someone who doesn't understand assembly language obviously wouldn't be able to find such a bug as quickly.

The local menu of the CPU view allows you to change the way items are displayed, look at a different address, go back to the current instruction pointer (EIP), search, go back to the source code, and so on. You can also pick the thread context in which to view the CPU information.

The Memory Dump pane enables you to view the contents of any range of memory. There are many ways in which it can be viewed—as `Byte`, `Word`, `DWORD`, `QWORD`, `Single`, `Double`, or `Extended`. You can search memory for a sequence of bytes as well as modify the data or follow it as code or data pointers.

The Register and Flags panes are pretty straightforward. All the CPU registers and flags are displayed here and can be modified.

The Stack pane gives you a stack-based view of the memory that's used for your program stack. In this pane, you can change values of data on the stack and follow addresses.

Summary

This chapter gives you some insight into the debugging process. It shows you the common problems you might run into while developing applications, and discusses the useful features of both the integrated and standalone debuggers. It's important to remember that debugging is as much a part of programming as is writing code. Your debugger can be one of your most powerful allies in writing clean code, so take the time to know it well.

In the next part of the book, you'll move into the realm of component-based development with COM and VCL components.