Turbo Debugger® for Windows

Version 3.1

User's Guide

BORLAND INTERNATIONAL, INC. 1800 GREEN HILLS ROAD
P.O. BOX 660001, SCOTTS VALLEY, CA 95067-0001

R1

10 9 8 7 6 5 4 3 2 1

C O N T E N T S

T A B L E S

F I G U R E S

I N T R O D U C T I O N

Turbo Debugger for Windows (TDW) is a state-of-the-art, source-level debugger designed to work with Borland's Turbo Pascal for Windows.

TDW enables you to debug applications you've written for Microsoft Windows, Version 3.0 and higher. It is a Windows application that runs on the same machine as the Windows program you're debugging. TDW switches between its own text-mode screens and your application's screens as you step through your application's code.

Multiple, overlapping windows, a combination of pull-down and pop-up menus, and mouse support provide a fast, interactive environment. An online context-sensitive help system provides you with help during all phases of operation.

Here are just some of TDW's features:

- debugging of Microsoft Windows applications
- full Turbo Pascal and assembler expression evaluation
- configurable screen layout
- assembler/CPU access when needed
- powerful breakpoint and logging facility
- back tracing
- OOP full support for object-oriented programming in Turbo Pascal for Windows
- operates in character mode

New features and changes for version 3.1

TDW 3.1 has the following enhancements over TDW 2.5:

- The Clipboard lets you copy from windows and paste either into text entry boxes on dialog boxes or into other windows. This feature is described on page 37.
- There are new breakpoint features (see Chapter 7) that let you
  - set multiple conditions and actions on a breakpoint
  - set and remove breakpoints in groups
  - set and remove breakpoints on all functions or procedures in a module
  - set and remove breakpoints on all methods in an object type
- International sort orders are supported through the Windows Language setting. You turn this feature on by using the configuration program TDWINST.EXE (see the file UTILS.TDW).
- The CPU window has a new pane that shows protected mode selectors and lets you look at the contents of memory locations referenced by these selectors (see page 181).
- The device driver TDDEBUG.386 provides support for the *Ctrl-Alt-SysRq* interrupt key combination. In addition, this device driver supports the hardware debugging registers of the Intel 80386 processor (and higher). See page 11 for TDDEBUG.386 installation information. See page 123 and the online file HDWDEBUG.TDW for information on hardware debugging.
- Debugging of DLLs is faster now that TDW simultaneously loads both the application's symbol table and the symbol table of any DLL you explicitly load or whose code you step into (see page 141).
- Extended graphics mode support is improved.

Hardware and software requirements

TDW requires or supports the following hardware and software:

- TDW has the same hardware and system software requirements as Turbo Pascal for Windows.
- TDW supports the following graphics modes and adapters: CGA, EGA, VGA, Hercules monochrome-graphics, Super VGA

(SVGA), and 8514. You can use standard drivers with everything except SVGA and 8514.

See the README file for more information on TDW.INI and the latest list of video drivers.

TDW supports SVGA and 8514 adapters through video support DLLs distributed with TDW. These DLLs support various adapter cards. You specify a video support DLL in TDW.INI, a file that's created in your WINDOWS subdirectory when you install TDW. To use an SVGA or 8514 DLL with TDW, copy it to the directory where TDW.EXE resides, then modify the VideoDLL line in your TDW.INI file to read:

```
VideoDLL = VGADRIVR.DLL
```

where *VGADRIVR.DLL* is the name of the required SVGA DLL.

See page 5 to find out how to contact Borland Technical Support.

If you can't find a DLL that matches your SVGA video adapter, contact Borland Technical Support to see if you can get one.

- TDW provides its own DLL, TDWIN.DLL, to support Windows debugging on both Windows 3.0 and Windows 3.1. This DLL replaces WINDEBUG.DLL and is copied to your hard disk during installation. The installation program also puts an entry in the TDW.INI file indicating where this DLL is to be found. This entry, *DebuggerDLL,* is in the [TurboDebugger] section of TDW.INI.

  For example, if TDWIN.DLL were installed in the C:\WINDOWS3.1 directory, you would see the following in TDW.INI:

  ```
  [TurboDebugger]
  DebuggerDLL=c:\windows3.1\tdwin.dll
  ```

- To use TDW, you must have Turbo Pascal for Windows. You must already have compiled your source code into an executable (.EXE or .DLL) file with full debugging information turned on.
- When you run TDW, you'll need the .EXE file, any DLLs you've written for it, and the original source files for both. The .EXE and any .DLL files must be in the same directory. TDW searches for source files first in the directory where the compiler found them when it compiled, second in the directory specified in the Options | Path for Source command, third in the current directory, and fourth in the directory the .EXE or .DLL file is in.

A note on terminology

For convenience and brevity, we use two terms in this manual in slightly more generic ways than usual. These terms are *module* and *argument.*

Module Refers to what is usually called a module in C++ and assembler, but also to what is called a *unit* in Pascal.

Argument Is used interchangeably with *parameter* in this manual. This applies to references to command-line arguments (or parameters), as well as to arguments (or parameters) passed to functions.

What's in the manual

Here is a brief synopsis of the chapters and appendixes in this manual:

Chapter 1: Getting started describes the contents of the distribution disk and tells you how to load TDW files into your system. It also gives you advice on which chapter to go to next, depending on your level of expertise.

Chapter 2: TDW basics explains the TDW environment, menus, and windows, and shows you how to respond to prompts and error messages.

Chapter 3: A quick example leads you through a sample session—using a Pascal program—that demonstrates many of the powerful capabilities of TDW.

Chapter 4: Starting TDW shows how to run the debugger from the command line, when to use command-line options, and how to record commonly used settings in configuration files.

Chapter 5: Controlling program execution demonstrates the various ways of starting and stopping your program, as well as how to restart a session or replay the last session.

Chapter 6: Examining and modifying data explains the unique capabilities TDW has for examining and changing data inside your program.

Chapter 7: Breakpoints introduces the concept of actions, and how they encompass the behavior of what are sometimes referred to as breakpoints, watchpoints, and tracepoints. Both conditional and unconditional actions are explained, as well as the various things that can happen when an action is triggered.

Chapter 8: Examining files describes how to examine program source files, as well as how to examine arbitrary disk files, either as text or binary data.

Chapter 9: Expressions describes the syntax of Pascal and assembler expressions accepted by the debugger, as well as the format control characters used to modify how an expression's value is displayed.

OOP

Chapter 10 Object-oriented debugging explains the debugger's special features that let you examine objects in Turbo Pascal for Windows.

Chapter 11: Using Windows debugging features describes how to use the TDW features that support debugging of Windows applications.

Chapter 12: Assembler-level debugging describes the CPU window. Additional information about this window and about assembler-level debugging is in the file ASMDEBUG.TDW.

Chapter 13: Debugging a standard Pascal program is an introduction to strategies for effective debugging of your programs.

OOP

Chapter 14: Debugging an ObjectWindows application leads you through a debugging session on a sample Windows program written using the ObjectWindows class library.

Appendix A: Error and information messages lists all the TDW prompts and error messages that can occur, with suggestions on how to respond to them.

How to contact Borland

Borland offers a variety of services to answer your questions about this product. Be sure to send in the registration card; registered owners are entitled to technical support and may receive information on upgrades and supplementary products.

Resources in your package

This product contains many resources to help you:

- The manuals provide information on every aspect of the program. Use them as your main information source.
- While using the program, you can press *F1* for help.
- Some common questions are answered in the file HELPME!.TDW, located in the DOC subdirectory of your language compiler directory, and the README file, located in the main language compiler directory.

Borland resources

Borland Technical Support publishes technical information sheets on a variety of topics and is available to answer your questions.

800-822-4269 (voice) TechFax

TechFax is a 24-hour automated service that sends free technical information to your fax machine. You can use your Touch Tone phone to request up to three documents per call.

408-439-9096 (modem) File Download BBS 2400 Baud

The Borland File Download BBS has sample files, applications, and technical information you can download with your modem. No special setup is required.

Subscribers to the CompuServe, GEnie, or BIX information services can receive technical support by modem. Use the commands in the following table to contact Borland while accessing an information service.

Online information services

Service	**Command**
CompuServe	`GO BORLAND`
BIX	`JOIN BORLAND`
GEnie	`BORLAND`

Address electronic messages to Sysop or All. Don't include your serial number; messages are in public view unless sent by a service's private mail system. Include as much information on the question as possible; the support staff will reply to the message within one working day.

408-438-5300 (voice) Technical Support 6 a.m. to 5 p.m. PST

Borland Technical Support is available weekdays from 6:00 a.m. to 5:00 p.m. Pacific time to answer any technical questions you have about Borland products. Please call from a telephone near your computer, and have the program running. Keep the following information handy to help process your call:

- product name, serial number, and version number
- the brand and model of any hardware in your system
- operating system and version number (use the DOS command `VER` to find the version number)
- contents of your AUTOEXEC.BAT and CONFIG.SYS files (located in the root directory (\) of your computer's boot disk)
- the contents of your WIN.INI and SYSTEM.INI files (located in your Windows directory) for TDW questions
- a daytime phone number where you can be contacted
- if the call concerns a problem, the steps to reproduce the problem

408-438-5300 (voice) Customer Service 7 a.m. to 5 p.m. PST

Borland Customer Service is available weekdays from 7:00 a.m. to 5:00 p.m. Pacific Time to answer any nontechnical questions you have about Borland products, including pricing information, upgrades, and order status.

C H A P T E R

1

Getting started

See the FILELIST.DOC file for information about the online files that document subjects not covered in this manual.

Turbo Debugger for Windows is part of the Turbo Pascal for Windows package, which consists of a set of distribution disks, the *Turbo Debugger for Windows User's Guide* (this manual), and the Turbo Pascal for Windows manuals. The distribution disks contain all the programs, files, and utilities needed to debug programs written in Turbo Pascal for Windows.

The distribution disks

When you install Turbo Pascal for Windows on your system, files from the distribution disks, including the TDW files, are copied to your hard disk. Just run INSTALL.EXE, the easy-to-use installation program on your distribution disks.

For a list of the files on the distribution disks, see the FILELIST.DOC file on the installation disk.

Online text files

There are a number of online files the installation program puts on your hard disk. The two you should definitely look at are README and FILELIST.DOC. They are accessible on the disk labeled "Installation Disk," and are also copied to your main language directory.

Additional files that provide information not found in the manual are HELPME!.TDW (commonly asked questions about TDW), ASMDEBUG.TDW (debugging of Assembler programs), UTILS.TDW (descriptions of utilities), and HDWDEBUG.TDW (hardware debugging). There also might be a file called MANUAL.TDW describing corrections to this user's guide. All these files by default are installed in the DOC subdirectory of your main language directory.

README

You can use the Turbo Pascal for Windows editor or Windows Notepad to access the README file.

It's very important that you take the time to look at the README file before you do anything else with TDW. This file contains last-minute information that might not be in the manual.

HELPME!.TDW

Your installation disk also contains a file called HELPME!.TDW, which contains answers to problems that users commonly run into. Consult it if you find yourself having difficulties. The HELPME!.TDW file discusses:

- the syntactic and parsing differences between TDW and Turbo Pascal for Windows
- debugging multi-language programs with TDW
- common questions about using TDW with Windows

ASMDEBUG.TDW

This file contains information on debugging Turbo Assembler programs. You might also find the information in this file helpful for debugging your inline assembler code.

UTILS.TDW

This file describes the command-line utilities included with TDW. These utility programs are TDSTRIP, TDUMP, and TDWINST.

Here's a brief description of each of the TDW utilities:

Use TDSTRIP to prepare .COM files for debugging.

- TDSTRIP.EXE lets you strip the debugging information (the *symbol table*) from your programs without relinking.

  A typical use of this utility is to create a .TDS file to use in debugging a .COM file. Because a .COM file you produce with a compiler has no symbol table information in it, you can debug it only by doing the following:

Compile the source code, with debugging information turned on, into a single-segment .EXE file, then run TDSTRIP on the .EXE. If the .EXE can be converted to a .COM file, TDSTRIP produces a .TDS file and a .COM file. You can now debug the .COM file by using the .TDS file with it.

- TDUMP.EXE displays the contents of object modules and .EXE files in a readable format.
- TDWINST.EXE lets you customize TDW. Using this utility, you can permanently set things like the display options and screen colors.

For a list of all the command-line options available for the TDW utility programs TDSTRIP.EXE and TDUMP.EXE, just type the program name and press *Enter*. For example, to see the command-line options for TDUMP.EXE, you'd type

```
TDUMP
```

To see the command-line options for TDWINST.EXE, type the program name and use the –? or –h option, then press *Enter*. For example, you could type

```
TDWINST -?
```

Installing TDW

The INSTALL.EXE program for Turbo Pascal for Windows also installs TDW. It creates a program group in the Windows program manager and creates icons for Turbo Pascal for Windows and TDW. See the README file for general installation information.

Installing TDDEBUG.386

There's a file on your installation disks, TDDEBUG.386, that provides the same functionality as the Windows SDK file WINDEBUG.386. In addition, it provides support for the hardware debugging registers of Intel 80386 (and higher) processors.

The installation program should copy this file to your hard disk and alter your Windows SYSTEM.INI file so that Windows loads

TDDEBUG.386 instead of WINDEBUG.386. If the installation program can't complete this task for you, it tells you. You then have to do it by hand, as follows:

1. The installation program will have copied TDDEBUG.386 from the installation disks to your hard disk. The standard directory for this file is C:\TPW. If you move the file to another directory, substitute that directory in the instructions.
2. With an editor, open the Windows SYSTEM.INI file, search for *[386enh]*, and add the following line to the 386enh section:

   ```
   device=C:\TPW\tddebug.386
   ```
3. If there's a line in the 386enh section that loads WINDEBUG.386, either comment the line out with a semicolon or delete it altogether. (You can't have both TDDEBUG.386 and WINDEBUG.386 loaded at the same time.)

   For example, if you load WINDEBUG.386 from the C:\WINDOWS directory, the commented-out line would be

   ```
   ;device=c:\windows\windebug.386
   ```

TDW.INI

TDW has its own initialization file, TDW.INI. This file has settings for the video driver and the location of the Windows-debugging DLL TDWIN.DLL. TDW.INI is introduced on page 3 and is described fully in the README file.

The installation program puts a copy of TDW.INI in the main Windows directory. In this copy of TDW.INI, the video driver setting (VideoDLL) is blank, and the DebuggerDLL setting indicates the path to TDWIN.DLL.

Hardware debugging

You can use the debugging registers of the Intel 80386 (and higher) processor to debug a Windows program. To use these registers, you must load TDDEBUG.386 when you start Windows (see the previous section).

See the online doc file HDWDEBUG.TDW for more information on debugging Windows programs using hardware debugging registers.

Where to now?

Now you can start learning about TDW. Since this *User's Guide* is written for three types of users, different chapters of the manual might appeal to you. The following road map will guide you.

Programmers learning Turbo Pascal

If you're just starting to learn Turbo Pascal, you'll want to be able to create small programs using it before you learn about the debugger. After you've gained a working knowledge of the language, work your way through Chapter 3, "A quick example," for a speedy tour of the major functions of TDW. There you'll learn enough about the features you need to debug your first program; you'll find out about the debugger's more sophisticated capabilities in later chapters.

Turbo Pascal pros; Turbo Debugger novices

If you're an experienced Turbo Pascal programmer but you're unfamiliar with Turbo Debugger, you can learn about the features of the TDW environment by reading Chapter 2, "TDW basics." If it suits your style, you can then work through the tutorial in Chapter 3, or, if you prefer, move straight on to Chapter 4, "Starting TDW."

Programmers experienced with Turbo Debugger

If you've used Turbo Debugger in the past, you're probably already familiar with TDW's standard features. In that case, you can go directly to Chapter 11, "Using Windows debugging features," which discusses the features of TDW that support Windows debugging. Another chapter you'll find helpful is Chapter 14, "Debugging an ObjectWindows application," which takes you through a debugging session on a Windows application written using the ObjectWindows library.

C H A P T E R

2

TDW basics

Debugging is the process of finding and correcting errors ("bugs") in your programs. It's not unusual to spend more time on finding and fixing bugs in your program than on writing the program in the first place. Debugging is not an exact science; the best debugging tool you have is your own "feel" for where a program has gone wrong. Nonetheless, you can always profit from a systematic method of debugging.

The debugging process can be broadly divided into four steps:

1. realizing you have a bug
2. finding where the bug is
3. finding the cause of the bug
4. fixing the bug

Is there a bug?

The first step can be really obvious. The computer freezes up (or *hangs*) whenever you run it. Or perhaps it crashes in a shower of meaningless characters. Sometimes, however, the presence of a bug is not so obvious. The program might work fine until you enter a certain number (like 0 or a negative number) or until you examine the output closely. Only then do you notice that the result is off by a factor of .2 or that the middle initials in a list of names are wrong.

Where is it?

The second step is sometimes the hardest: isolating where the error occurs. Since you can't keep the entire program in your head at one time (unless it's a very small program), your best approach is to divide and conquer—break up the program into parts and debug them separately. Structured programming is perfect for this type of debugging.

What is it?

The third step, finding the cause of the error, is probably the second-hardest part of debugging. Once you've discovered where the bug is, it's usually somewhat easier to find out why the program is misbehaving. For example, if you've determined the error is in a procedure called *PrintNames*, you only have to examine the lines of that procedure instead of the entire program. Even so, the error can be elusive and you might need to experiment a bit before you succeed in tracking it down.

Fixing it

The final step is fixing the error. Armed with your knowledge of the programming language and knowing where the error is, you can squash the bug. Now you run the program again, wait for the next error to show up, and start the debugging process again.

See Chapter 13 for a more detailed discussion of the debugging process.

Many times this four-step process is accomplished when you are writing the program itself. Syntax errors, for example, prevent your programs from compiling until they're corrected. Turbo Pascal for Windows has a built-in syntax checker that informs you of these errors and lets you fix them on the spot.

But other errors are more insidious and subtle. They lie in wait until you enter a negative number, or they're so elusive you're stymied. That's where TDW comes in.

What TDW can do for you

With TDW, you have access to a much more powerful debugger than could exist in your language compiler.

You can use TDW with any program written in Turbo Pascal for Windows. TDW runs in character mode and allows you to switch to your application running under Windows.

TDW helps with the two hardest parts of the debugging process: finding where the error is and finding the cause of the error. It does this by controlling program execution so you can examine the state of the program at any given spot. You can even test new values in variables to see how they affect your program. With TDW, you can perform *tracing*, *back tracing*, *stepping*, *viewing*, *inspecting*, *changing*, and *watching*.

Tracing	Executing your program one line at a time.
Back tracing	Tracing backward through your executed code, reversing the execution as you go.
Stepping	Executing your program one line at a time, but stepping over any routines or function calls. If you're sure your routines and functions are error-free, stepping over them speeds up debugging.
Viewing	Opening a special TDW window to see the state of your program from various perspectives: variables, their values, breakpoints, the contents of the stack, a log, a data file, a source file, CPU code, memory, registers, numeric coprocessor information, object or class hierarchies, execution history, or program output.
Inspecting	Delving deeper into the workings of your program by examining the contents of complicated data structures like arrays.
Changing	Replacing the current value of a variable, either globally or locally, with a value you specify.
Watching	Isolating program variables and keeping track of their changing values as the program runs.

You can use these powerful tools to dissect your program into discrete chunks, confirming that one chunk works before moving to the next. In this way, you can burrow through the program, no matter how large or complicated, until you find where that bug is hiding. Maybe you'll find there's a function that inadvertently reassigns a value to a variable, or maybe the program gets stuck in an endless loop, or maybe it gets pulled into an unfortunate recursion. Whatever the problem, TDW helps you find where it is and what's at fault.

OOP

TDW lets you debug object-oriented Pascal programs. It is smart about objects, and it correctly handles late binding of virtual methods so that it always executes and displays the correct code.

What TDW won't do

With all the features built into TDW, you might be thinking that it's got it all. In truth, there are at least three things TDW *won't* do for you.

- TDW cannot recompile your program for you. You need Turbo Pascal for Windows to do that.
- TDW doesn't run in graphics mode under Windows, but rather runs in character mode.
- TDW does not take the place of thinking. When you're debugging a program, your greatest asset is using your head. TDW is a powerful tool, but if you use it mindlessly, it's unlikely to save you time or effort.

How TDW does it

Here's the really good news: TDW gives you all this power and sophistication, and at the same time it's easy—even intuitive—to use.

TDW accomplishes this blend of power and ease by offering an integrated debugging environment. The next section examines the advantages of this environment.

The TDW advantage

Once you start using TDW, we think you'll be unable to get along without it. TDW has been especially designed to be as easy and convenient as possible. To this end, TDW offers you these features:

- Convenient and logical global menus.
- Context-sensitive local menus throughout the product, which practically do away with memorizing and typing commands.
- Dialog boxes in which you can choose, set, and toggle options and type in information.

- When you need to type, TDW keeps a *history list* of the text you've typed in similar situations. You can choose text from the history list, edit the text, or type in new text.
- Full macro control to speed up series of commands and keystrokes.
- Copying and pasting between windows and dialog boxes.
- Convenient, complete window management.
- Mouse support.
- Access to several types of online help.
- Reverse execution.
- Single and dual monitor support.

The rest of this chapter discusses these features of the TDW environment.

Menus and dialog boxes

As with other Borland products, TDW has a convenient global menu system accessible from a menu bar running along the top of the screen. This menu system is always available except when a dialog box is active.

A *pull-down menu* is available for each item on the menu bar. Through the pull-down menus, you can

- execute a command.
- open a *pop-up menu*. Pop-up menus appear when you choose a menu item that is followed by a menu icon (▸).
- open a *dialog box*. Dialog boxes appear when you choose a menu item that is followed by an ellipsis (...).

Using the menus

There are four ways you can open the menus on the menu bar:

Getting in

- Press *F10*, use → or ← to go to the desired menu, and press *Enter*.
- Press *F10*, then press the first letter of the menu name (*F, E, V, R, B, D, O, W, H*, or *Spacebar* for the System menu).
- Press *Alt* plus the first letter of any menu bar command (*F, E, V, R, B, D, O, W, H*, or *Spacebar* for the System menu). For example, wherever you are in the system, *Alt-F* takes you to the File menu. The ≡ (System) menu opens with *Alt-Spacebar*.
- Click the menu bar command with the mouse.

Once you are in the global menu system, here is how you move around in it:

Getting around

- Use → and ← to move from one pull-down menu to another. (For example, when you are in the File menu, pressing → takes you to the Edit menu.)
- Use ↑ and ↓ to scroll through the commands in a specific menu.
- Use *Home* and *End* to go to the first and last menu items, respectively.
- Highlight a menu command and press *Enter* to move to a lower-level (pop-up) menu or dialog box.
- Click the mouse on a command to move to a lower-level (pop-up) menu or dialog box.

This is how you get out of a menu or the menu system:

Getting out

- Press *Esc* to exit a lower-level menu and return to the previous menu.
- Press *Esc* in a pull-down menu to leave the menu system and return to the active window.
- Press *F10* in any menu (but *not* in a dialog box) to exit the menu.
- Click a window with the mouse to leave the menu system and go to that window.

Some menu commands have a shortcut *hot key* that you press to execute them. The hot key appears in the menu to the right of these commands.

Figure 2.10 shows the complete pull-down menu tree for TDW.

Dialog boxes

Many of TDW's command options are available to you in *dialog boxes*. A dialog box contains one or more of the following items:

Table 2.1
What goes in a dialog box

The hot key for the OK button is Alt-K.

	Item	**What it looks like, what it does**
	Buttons	Buttons are "shadowed" text (on monochrome systems they appear in reverse video). If you choose a button, TDW carries out the related action immediately. Get out of a dialog box by pressing the button marked OK to confirm your choices, or Cancel to cancel them. Dialog boxes also contain a Help button that brings up online help.
[X]	**Check boxes**	A check box is an on/off toggle. Choose it to turn the option on or off. When a check box option is turned on, an X appears in brackets: `[X]`.
( ) (•) ( )	**Radio buttons**	Radio buttons offer a set of toggles, but the choices are mutually exclusive: you can choose only one radio button in a set at a time. When you do, a bullet appears between the parentheses, as follows: `(•)`.
_	**Input boxes**	An input box prompts you to type in a string (the name of a file, for example). An input box often has a *history list* associated with it (see "History lessons" in this chapter for more on history lists).
THISFILE.EXE THATFILE.EXE TOTHERFL.EXE	**List boxes**	A list box contains a list of items you can choose from (for example, a list of possible files to open).

You navigate around dialog boxes by pressing *Tab* and *Shift-Tab*. Within sets of radio buttons, use the arrow keys to change the settings. To choose a button, tab to it and press *Enter*.

If you have a mouse, it is even easier to get around in a dialog box. Just click the item you want to choose. To cancel the dialog box, click the close box in the upper left corner.

You can also choose items in a dialog box by pressing their hot key, the highlighted letter in each command.

Knowing where you are

In addition to the convenient system of Borland pull-down menus, the TDW advantage consists of a powerful feature that lessens confusion by actually reducing the number of menus.

To understand this feature, you must realize that first and foremost, TDW is context-sensitive. That means it keeps tabs on exactly which window you have open, what text is selected, and which subdivision, or *pane*, of the window your cursor is in. In other words, it knows precisely what you're looking at and where the cursor is when you choose a command. And it uses this information when it responds. Let's take an example to illustrate.

Suppose your program has a line like this:

```
MyCounter[TheGrade] := MyCounter[TheGrade] + 1;
```

As you'll discover when you work with TDW, getting information on data structures is easy; all you do is press *Ctrl-I*, the hot key that opens an Inspector window, to *inspect* it. When the cursor is at *MyCounter*, TDW shows you information on the contents of the entire array variable. But if you were to select (that is, highlight) the whole array name and the index and then press *Ctrl-I*, TDW knows that you want to inspect one component and shows you only that component.

You can tunnel down to finer and finer program detail in this way. Pressing *Ctrl-I* on a highlighted component while you're already inspecting an array gives you a look at that component.

This sort of context-sensitivity makes TDW extremely easy to use. It saves you the trouble of memorizing and typing complicated strings of menu commands or arcane command-line switches. You simply move to the item you want to examine (or select it using the *Ins* key or drag over it with the mouse), and then invoke the command (*Ctrl-I* for Inspect, for example).

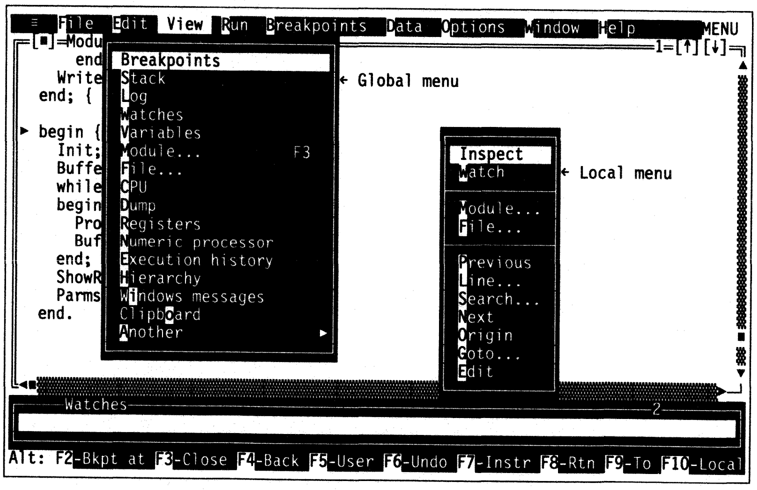
This context-sensitivity, which makes life easy for the user, also makes the task of documenting commands difficult. This is because *Ctrl-I*, for example, in TDW does not have a *single* result; instead, *the outcome of a command depends on where your cursor is or what text is selected.*

Local menus

Another aspect of TDW's context-sensitivity is in its use of *local menus* specific to different windows or panes within windows.

Local menus in TDW are tailored to the particular window or pane you are in. It's important not to confuse them with global menus. Here is a composite screen shot of both kinds of menus (when you're actually working in TDW, however, you could never have both types of menus showing at the same time):

Figure 2.1
Global and local menus



The following table compares global and local menus:

Table 2.2
Global and local menu operations

Global menus	Local menus
You access a global menu by pressing *F10* and using the arrow keys or typing the first letter of the the menu name.	You call up a local menu by pressing *Alt-F10* or by clicking the right button on your mouse.
A global menu is always available from the menu bar, visible at the top of the screen.	The placement and contents of the menu depend on which window or pane you are in and where your cursor is.
The contents of a global menu never change.	Contents can vary from one local menu to another. Even so, many of the local commands appear in almost all of the local menus, so that there's a predictable core of commands from one to another. The *results* of like-named commands can be different, however, depending on the context.
Some of the menu commands have hot key shortcuts that are available from anywhere in TDW.	Every command on a local menu has a hot key shortcut consisting of *Ctrl* plus the highlighted letter in the command. A hot key is available only when the associated window or local menu is active.

Table 2.2: Global and local menu operations (continued)

	Because of this arrangement, a hot key, say *Ctrl-S*, might mean one thing in one context but quite another in a different context. (A core of commands, however, is still consistent across the local menus. For example, the Goto command and the Search command always do the same thing, even when they are invoked from different panes.)

From a user's standpoint, local menus are a great convenience. All possible command choices relevant to the moment are laid out at a glance. This feature helps you avoid choosing inappropriate commands and keeps the menus small and uncluttered.

History lessons

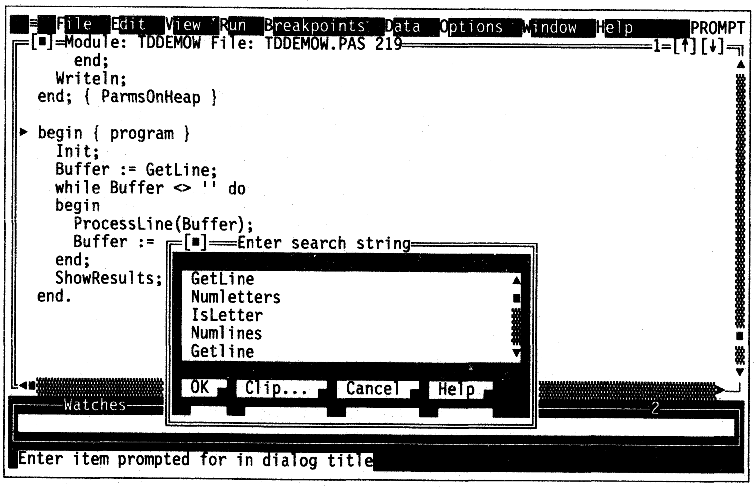
Menus and context-sensitivity comprise just two aspects of the convenient environment of TDW. Another habit-forming feature is the *history list*.

Conforming to the philosophy that the user shouldn't have to type more than absolutely necessary, TDW remembers whatever you enter into input boxes and displays that text whenever you call up the box again.

For example, to search for the function called *MyPercentage*, you have to type in all or part of that word. Then suppose you want to search for a variable called *ReturnOnInvestment*. When you see the dialog box this time, you'll notice that *MyPercentage* appears in the input box. When you search for another text string, both previously entered strings appear in the input box. The list keeps growing as you continue to use the Search command.

The search input box might look like this:

Figure 2.2
A history list in an input box



File Edit View Run Breakpoints Data Options Window Help PROMPT
Watches
OK
Clip...
Cancel
Help

```
[■]=Module: TDDEMOW File: TDDEMOW.PAS 219                    1=[↑][↓]
    end;
  Writeln;
end; { ParmsOnHeap }

► begin { program }
  Init;
  Buffer := GetLine;
  while Buffer <> '' do
  begin
    ProcessLine(Buffer);
    Buffer :=  [■]==Enter search string==
  end;
  ShowResults;   GetLine
end.             Numletters
                 IsLetter
                 Numlines
                 Getline

Enter item prompted for in dialog title
```

The first item in a search list is always the word the cursor is on in the Module window.

You can use this history list as a shortcut to typing by using the arrow keys to select any previous entry, then pressing *Enter* to start the search. If you have a mouse, you can also use the scroll bar to scroll to the entry you want. If you use an unaltered entry from the history list, that entry is copied to the top of the list.

You can also edit entries (use the arrow keys to insert the cursor in the highlighted text, then edit as usual, using *Del* or *Backspace*). For example, you can select *MyPercentage* and change it to *HisPercentage*, instead of typing in the entire text. If you start to type a new item when an entry is highlighted, you will overwrite the highlighted item.

A history list lists the last ten responses unless you've used TDWINST to configure TDW otherwise. (The TDWINST program is described in the online text file UTILS.TDW.)

TDW keeps a separate history list for most input boxes. That way, the text you enter to do a search does not clutter up the box for, say, going to a particular label or line number.

Automatic name completion

Whenever you are prompted for text entry in an input box, you can type in just part of a symbol name in your program, then press *Ctrl-N*.

When the word `READY...` appears in the upper right corner of the screen with three dots after it, it means the symbol table is being sorted. *Ctrl-N* won't work until the three dots go away, indicating that the symbol table is available for name completion.

Ctrl N

- If you have typed enough of a name to uniquely identify it, TDW simply fills in the rest of it.
- If the name you have typed so far is not the beginning of any known symbol name, nothing happens.
- If what you have typed matches the beginning of more than one symbol name, a list of matching names is presented for you to pick the one you want.

Incremental matching

TDW also lets you use *incremental matching* to find entries in a dialog box list of file and directory names. Start typing the name of the file or directory; if the file is available from the names at or below the current position in the list box, the highlight bar moves to the name as soon as you have typed enough characters to identify it uniquely. Then all you have to do is choose the OK button.

Making macros

Whenever you find yourself repeating a series of steps, say to yourself, "Shouldn't I be using a macro for this?"

Macros are simply hot keys you define to perform a series of commands and other keystrokes.

You can assign any series of TDW commands and keystrokes to a single key, for playback whenever you want.

See page 67 in Chapter 4 for an explanation of how to define macros.

Window shopping

TDW displays all information and data in menus (local and global), dialog boxes (which you use to set options and enter information), and windows. There are many types of windows; a window's type depends on what sort of information it holds. You open and close all windows using menu commands (or hot key shortcuts for those commands). Most of TDW's windows come from the View menu, which lists fifteen types of windows. Another class of window, called the Inspector window, is opened by choosing either Data | Inspect or Inspect from a local menu.

Windows from the View menu

```
Breakpoints
Stack
Log
Watches
Variables
Module...        F3
File...
CPU
Dump
Registers
Numeric processor
Execution history
Hierarchy
Windows messages
Clipboard
Another           ▸
```

To the left is a list of the fifteen types of windows you can open from the View menu.

Once you have opened one or more of these windows, you can move, resize, close, and otherwise manage them with commands from the Window and ≡ (System) menus, which are discussed later in this chapter in the section "Working with windows."

Breakpoints window

See Chapter 7 for a complete description of this type of window and how breakpoints work.

Displays the breakpoints you have set. A breakpoint defines a location in your program where execution stops so you can examine the program's status. The left pane lists the position of every breakpoint (or indicates that it is global), and the right pane indicates the conditions under which the currently highlighted breakpoint executes.

Use this window to modify, delete, or add breakpoints.

Stack window

Chapter 5 provides more information on the Stack window.

Displays the current state of the stack, with the function called first on the bottom and all subsequently called functions on top, in the order in which they were called.

You can bring up and examine the source code of any function in the stack by highlighting it and pressing *Ctrl-I.*

By highlighting a function name in the stack and pressing *Ctrl-L,* you open a Variables window displaying variables global to the program, variables local to the function, and the arguments with which the function was called.

Log window

Chapter 7 tells you more about the Log window.

Displays the contents of the message log. The log contains a scrolling list of messages and information generated as you work in TDW. It tells you such things as why your program stopped, the results of breakpoints, and the contents of windows you saved in the log.

You can also use the log window to obtain information about memory usage, modules, and window messages for your Windows application.

This window lets you look into the past and see what led to the current state of affairs.

Watches window

See Chapter 6 for more about the Watches window.

Displays variables and expressions and their changing values as the program executes. You can add a variable to the window by pressing *Ctrl-W* when the cursor is on the variable in the Module window.

Variables window

Chapter 5 describes the Variables window in more detail.

Displays all the variables accessible from a given spot in your program. The upper pane has global variables; the lower pane shows variables local to the current function or module, if any.

This window is helpful when you want to find a function or variable that you know begins with, say, "abc," and you can't remember its exact name. You can look in the global Symbol pane and quickly find what you want.

Module window

Displays the program code for the module you're debugging and for any DLLs called from the module. You can move around inside the module or DLL and examine data and code by positioning the cursor on program variable names and issuing the appropriate local menu command.

Chapter 8 details the Module window and its commands.

You will probably spend more time in Module windows than in any other type, so take the time to learn about all the various local menu commands for this type of window.

You can also press *F3* to open a Module window.

File window

You can learn more about the File window in Chapter 8.

Displays the contents of a disk file. You can view the file either as raw hex bytes or as ASCII text, and you can search for specific text or byte sequences.

CPU window

Chapter 12 discusses the CPU window and assembler-level debugging.

Displays the current state of the central processing unit (CPU). This window has six panes: one that contains disassembled machine instructions, one that shows the contents of a selector, one that shows hex data bytes, one that displays a raw stack of hex words, one that lists the contents of the CPU registers, and one that indicates the state of the CPU flags.

The CPU window is useful when you want to watch the exact sequence of instructions that make up a line of source code or the bytes that comprise a data structure. If you know assembler code, this can help locate subtle bugs. You do not need to use this window to debug the majority of programs.

TDW sometimes opens a CPU window automatically if your program stops in Windows code or on an instruction in the middle of a line of source code.

Dump window

See Chapter 12, which discusses assembler debugging, for more on this window.

Displays a raw display of an area of memory. (This window is the same as the Data pane of a CPU window.) You can view the data as characters, hex bytes, words, doublewords, or any floating-point format. You can use this window to look at some raw data when you don't need to see the rest of the CPU state or to gain direct access to I/O ports. The local menu has commands to let you modify the displayed data, change the format in which you view the data, and manipulate blocks of data.

Registers window

Chapter 12, which discusses assembler debugging, has more information on this window.

Displays the contents of the CPU registers and flags. This window has two panes, which are the same as the registers pane and flags pane, respectively, of a CPU window. Use this window when you want to look at the contents of the registers but don't need to see the rest of the CPU state. You can change the value of any of the registers or flags through commands in the local menu.

Numeric Processor window

See the file ASMDEBUG.TDW for more information about using the Numeric Processor window.

Displays the current state of the numeric coprocessor. This window has three panes: one pane that shows the contents of the floating-point registers, one that shows the status flag values, and one that shows the control flag values.

This window can help you diagnose problems in programs that use floating-point numbers. You need to have a fair understanding of the inner workings of the numeric coprocessor in order to really reap the benefits of this window.

Execution History window

See Chapter 5 for more information on the Execution History window.

Displays source lines for your program, up to the last line executed. The window indicates

1. whether you are tracing or stepping
2. the line of source code for the instruction about to be executed
3. the line number of the source code

You can examine it or use it to rerun your program to a particular spot.

Hierarchy window

OOP

See Chapter 10 for more information about using the Hierarchy window.

Lists and displays a hierarchy tree of all object types used by the current program. The window has two panes: one for the object type list and the other for the object hierarchy tree. This window shows you the relationship of the objects used by the current module. By using this window's local menu commands, you can examine any object type's data fields and methods.

Windows Messages window

Chapter 11 explains how to use the Windows Messages feature.

Displays a list of messages passed between the windows in your Windows application. This window has three panes:

- The left pane shows which procedures or handles you're tracking messages for.
- The right pane shows the type of messages you're tracking.
- The bottom pane displays the messages being tracked.

Clipboard window

See page 37 for an explanation of how to use TD's Clipboard.

Displays the items that have been clipped into the Clipboard, showing you their types and letting you inspect or delete an item and freeze the value of any item in the Clipboard.

Duplicate windows

```
Module...
Dump
File...
```

You can also open duplicates of three types of windows—Dump, File, and Module—by choosing View | Another. This lets you keep track of several separate areas of assembly code, different files the program uses or generates, or several distinct program modules at once.

Don't be alarmed if TDW opens one of these windows all by itself. It will do this in some cases in response to a command.

User screen

The User screen shows your program's full output screen. The screen you see is exactly the same as the one you would see if your program was running directly under Windows and not under TDW.

Alt-F5 is the hot key that toggles between the environment and the User screen.

You can use this screen to check that your program is at the place in your code that you expect it to be, as well as to verify that it is displaying what you want on the screen. To switch to the User screen, choose Window | User Screen. After viewing the User screen, press any key to go back to the debugger screen.

Inspector windows

An Inspector window displays the current value of a selected variable. Open it by choosing Data | Inspect or Inspect from a local menu. Usually, you close this window by pressing *Esc* or clicking the close box with the mouse. If you've opened more than one Inspector window in succession, as often happens when you examine a complex data structure, you can remove all the Inspector windows by pressing *Alt-F3* or using the Window | Close command.

You can open an Inspector window to look at an array of items or at the contents of a variable or expression. The number of panes in the window depends on the nature of the data you are inspecting. An Inspector window adapts to the type of data being displayed. It can display not only simple scalars (*integer*, *char*, and so on), but also pointers, arrays, and records. Each type of data item is displayed in a way that closely mimics the way you're used to seeing it in your program's source code.

Although you create additional Inspector windows simply by choosing the Inspect command, you can create additional Module, File, or CPU windows only by choosing View | Another.

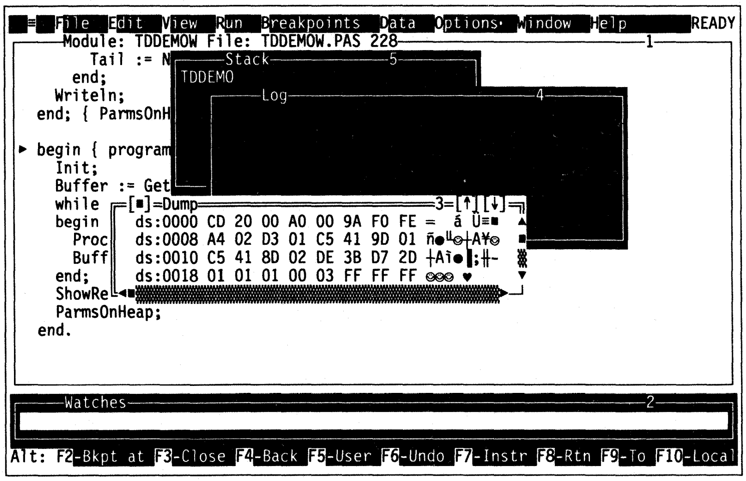
The active window

Even though you can have many windows *open* in TDW at the same time, only one window can be *active*. You can spot the active window by using the following criteria:

- The active window has a double outline around it, not a single line.
- The active window contains the cursor or highlight bar.
- If your windows are overlapping, the active window is the topmost one.

When you issue commands, enter text, or scroll, you affect only the active window, not any other windows that are open.

Figure 2.3
The active window has a double outline

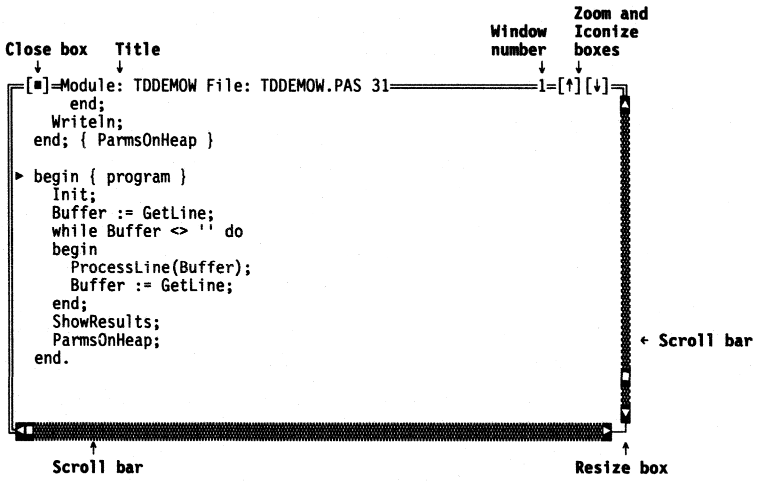


```
 ≡ File  Edit  View  Run  Breakpoints  Data  Options· Window  Help        READY
┌────Module: TDDEMOW File: TDDEMOW.PAS 228───────────────────────────1──────┐
      Tail := N┌───Stack───────────5─┐
    end;        │TDDEMO
  Writeln;      │  ┌───Log──────────────────4───┐
end; { ParmsOnH │  │
                │  │
▶ begin { program│ │
  Init;          │ │
  Buffer := Get  └─│
  while ╔[■]═Dump═════════════════════3═[↑][↓]═╗
  begin ║ ds:0000 CD 20 00 A0 00 9A F0 FE =  á Ü≡■   ▲
    Proc║ ds:0008 A4 02 D3 01 C5 41 9D 01 ñ●╙☺┼A¥☺  ■
    Buff║ ds:0010 C5 41 8D 02 DE 3B D7 2D ┼Aì●▐;╫-  ▓
  end;  ║ ds:0018 01 01 01 00 03 FF FF FF ☺☺☺ ♥     ▼
  ShowRe╚◄■▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓►┘
  ParmsOnHeap;
end.

┌────Watches─────────────────────────────────────────────────────────2──────┐
│                                                                            │
Alt: F2-Bkpt at F3-Close F4-Back F5-User F6-Undo F7-Instr F8-Rtn F9-To F10-Local
```

What's in a window

A window always has most or all of the following features, which give you information about it or let you do things to it:

Figure 2.4
A typical window



```
=[■]=Module: TDDEMOW File: TDDEMOW.PAS 31=============1=[↑][↓]=
      end;
    Writeln;
  end; { ParmsOnHeap }

► begin { program }
    Init;
    Buffer := GetLine;
    while Buffer <> '' do
    begin
      ProcessLine(Buffer);
      Buffer := GetLine;
    end;
    ShowResults;
    ParmsOnHeap;
  end.
```

- An outline (double if the window is active, single otherwise).
- A title, located at the left top.
- A scroll bar or bars on the right or bottom if the window opens on more information than it can hold at one time. You operate the scroll bars with the mouse:
  - Click the direction arrows at the ends of the bar to move one line or one character in the indicated direction.
  - Click the gray area in the middle of the bar to move one window size in the indicated direction.
  - Drag the scroll box to move as much as you want in the direction you want.
- A resize box in the lower right corner. Drag it with your mouse to make the window larger or smaller. If no scroll bar is present on the bottom or right side of a window, that side of the window border also activates window resizing.
- A window number in the upper right, reflecting the order in which the window was opened.
- A zoom box and iconize box in the upper right corner. The one on the left contains the zoom icon, the one on the right the iconize icon. Click these with your mouse to expand the window to full screen size, restore it to its original size, or iconize it. (When a window is zoomed to full size, only the unzoom box is available, and when it is iconized, only the zoom box is available.)

- A close box in the upper left corner. Click it with your mouse to close the window.

Working with windows

With all these different windows to work with, you will probably have several open onscreen at a time. TDW makes it easy for you to move from one window to another, move them around, pile them on top of one another, shrink them to get them out of your way, expand them to work in them more easily, and close them when you are through.

Press Alt-Spacebar to open the ≡ menu, or Alt-W to open the Window menu.

Most of the window-management commands are in the Window menu. You'll find a few more commands in the ≡ (System) menu, the menu marked with the ≡ icon at the far left of the menu bar.

Window hopping

Each window that you open is numbered in the upper right corner. Usually, the Module window is window 1 and the Watches window is window 2. Whatever window you open after that will be window 3, and so on.

This numbering system gives you a quick, easy means of moving from one window to another. You can make any of the first nine open windows the active window by pressing *Alt* in combination with the window number. If you press *Alt-2,* for example, to make the Watches window active, any commands you choose will affect that window and the items in it.

F6 is the hot key for the Window | Next Window.

You can also cycle through the windows onscreen by choosing Window | Next or pressing *F6.* This is handy if an open window's number is covered up so you don't know which number to press to make it active.

If you have a mouse, you can also activate a window by clicking it.

To see a list of all open windows, choose Window from the menu bar. The bottom half of the Window menu lists up to nine open windows from which you can make a selection. Just press the number of a window to make it the active one.

If you have more than nine windows open, the window list will include a Window Pick command; choose it to open a pop-up menu of all the windows open onscreen.

Tab and Shift-Tab are the hot keys for Window | Next Pane.

If a window has *panes*—areas of the window reserved for a specific type of data—you can move from one pane to another by choosing Window | Next Pane or pressing *Tab* or *Shift-Tab*.

You can also click the pane with the mouse.

The most pane-full window in TDW is the CPU window, which has six panes.

As you hop from pane to pane, you'll notice that a blinking cursor appears in some panes, and a highlight bar appears in others. If a cursor appears, you move around the text using standard keypad commands. (*PgUp*, *Ctrl-Home*, and *Ctrl-PgUp*, for example, move the cursor up one screen, to the top of pane, or to the top of the list, respectively.) If you've disabled shortcut keys, you can also use WordStar-like hot keys for moving around in the pane.

If there's a highlight bar in a pane instead of a cursor, you can still use standard cursor-movement keys to get around, but a couple of special keystrokes also apply. In alphabetical lists, for example, you can *select by typing*. As you type each letter, the highlight bar moves to the first item starting with the letters you've just typed. The position of the cursor in the highlighted item indicates how much of the name you have already typed. Once the highlight bar is on the desired item, your search is complete. This incremental matching or select by typing minimizes the number of characters you must type in order to choose an item from a list.

Once an item is selected (highlighted) from a list, you can press *Alt-F10* or *Ctrl-F10* or the right mouse button to display the local menu and choose a command relevant to the highlighted item. In many lists, you can also just press *Enter* once you have selected an item. This acts as a hot key to one of the commonly used local menu commands.

Finally, a number of panes let you start typing a new value or search string without choosing a command first. This usually applies to the most frequently used local menu command in a pane or window—like Goto in a Module window, Search in a File window, or Change in a Registers window.

Moving and resizing windows

When you open a new window in TDW, it appears near the current cursor location and has a default size suitable for the kind of window it is. If you find either the size or the location of the window inconvenient, you can use the Window | Size/Move command to adjust the size or location of the window.

Ctrl-F5 is the hot key for the Window | Size/Move command.

When you move or resize a window, your active window border changes to a single-line border. You can then use the arrow keys to move the window around or *Shift* with the arrow keys to change the size of the window onscreen. Press *Enter* when you're satisfied.

If you have a mouse, moving and resizing a window is even easier:

- Drag the resize box in the lower right corner to change the size of the window.
- Drag the title bar or any edge (but not the scroll bars) to move the window around.

F5 is the hot key for the Window | Zoom command.

If you want to enlarge or reduce a window quickly, choose Window | Zoom, or click the mouse on the zoom box or the iconize box (one or two boxes each containing an arrow) in the upper right corner of the window.

Finally, if you want to get a window out of the way temporarily but don't want to close it, make the window active, then choose Window | Iconize/Restore. The window will shrink to a tiny box (icon) with only its name, close box, and zoom box visible. To restore the window to its original form, make it active and choose Window | Iconize/Restore again, or click your mouse on the zoom box.

Closing and recovering windows

Alt-F3 is the hot key for Window | Close.

When you are through working in a window, you can close it by choosing Window | Close.

If you have a mouse, you can also click the Close box in the upper left corner of the window.

Alt-F6 is the hot key for Window | Undo Close.

If you close a window by mistake, you can recover it by choosing Window | Undo Close or by pressing *Alt-F6*. This works *only* for the last window you closed.

You can also restore your TDW screen to the layout it had when you first entered the program. Just choose ≡ (System) | Restore Standard.

Finally, if your program has overwritten your environment screen with output (because you turned off screen swapping), you can clean it up again with ≡ (System) | Repaint Desktop.

Saving your window layout

Use the Options | Save Options command to save a specific window configuration once you have the screen arranged the way you like. In the Save Configuration dialog box, tab to Layout and press *Spacebar* to toggle it on. In the Save To text box, indicate the configuration file to save to (TDCONFIG.TDW is the default), then press *Enter* or click OK to save the configuration.

If you save the configuration to TDCONFIG.TDW, the screen will appear with your chosen layout each time you start TDW. This configuration file is the only one loaded automatically when TDW is loaded. You can load other configurations with other names by using the Options | Restore Options command.

Copying and pasting

TDW has an extensive copy and paste feature called the Clipboard. With the Clipboard you can copy and paste between TDW windows and dialog boxes.

The items you copy into the Clipboard are dynamic; if an item has an associated value, the Clipboard keeps that value current as it changes in your program.

You can use the Ins key to mark multiple items in a list.

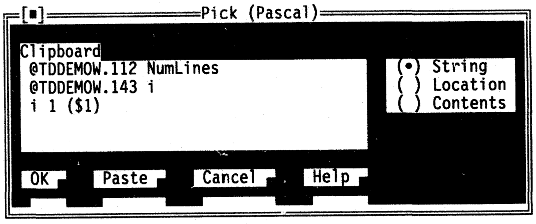
To copy an item into the Clipboard, position the cursor on the item or highlight it with the *Ins* key, then press *Shift-F3*. To paste something into a window or dialog box from the Clipboard, press *Shift-F4* or use the Clip button in the dialog box to bring up the Pick dialog box.

☞ You can paste into any dialog box prompt (any place in a dialog box where you can type text) by pressing *Shift-F4*, even if the dialog box doesn't have a Clip button. You can also paste into dialog box prompts with multiple fields.

The Pick dialog box

Pressing *Shift-F4* or the Clip button brings up a dialog box listing Clipboard contents and showing the categories you can use for pasting an item into the dialog box.

Figure 2.5
The Pick dialog box



```
[■]==============Pick (Pascal)===============
Clipboard
 @TDDEMOW.112 NumLines              (•) String
 @TDDEMOW.143 i                     ( ) Location
 i 1 ($1)                           ( ) Contents

 OK    Paste    Cancel    Help
```

This dialog box shows a scrolling list of items in the Clipboard and allows you to interpret the item to be pasted in up to three ways: as a string, as an address, or as contents of an address. The categories you can use in pasting the item depend on its type and its destination (discussed later).

For example, if you clip text from the Log window, it can be pasted only as a string. If you clip text from the Module window, it can be pasted elsewhere as a string or as an address, but not as contents. If you clip a variable from an Inspector window, it can be pasted as a string, a location, or as contents.

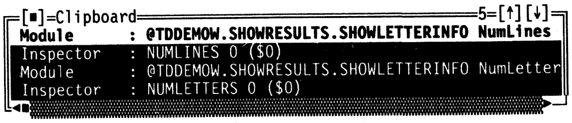
To paste an item into a dialog box, highlight the item, select the appropriate category, then either press *Enter* or the Paste button, depending on what effect you want to have on the dialog box.

- Pressing *Enter* simply pastes the item in and returns you to the dialog box.
- Pressing the Paste button both pastes the item in and passes an *Enter* to the dialog box, causing it to perform its function.

The Clipboard window

There's a View window that lets you see the contents of the Clipboard. Choosing View | Clipboard displays the Clipboard window, which lists all clipped items.

Figure 2.6
The Clipboard window



```
[■]=Clipboard===============================5=[↑][↓]=
Module      : @TDDEMOW.SHOWRESULTS.SHOWLETTERINFO NumLines
Inspector   : NUMLINES 0 ($0)
Module      : @TDDEMOW.SHOWRESULTS.SHOWLETTERINFO NumLetter
Inspector   : NUMLETTERS 0 ($0)
```

The leftmost field of this window describes the type of the entry, followed by a colon and the clipped item. If the clipped item is an expression from the Watches window, a variable from the Inspector window, or data, a register, or a flag from the CPU window, the item is followed by its value or values.

Clipboard item types

When you clip an item from a Window, Turbo Debugger assigns it a type to help you identify the source of the item. The following table shows the Clipboard types:

Table 2.3
Clipboard item types

Type	Description
String	A text string, like a marked block from the File window
Module	A module context, including a source code position, like a variable from the Module window
File	A position in a file (in the File window) that isn't a module in the program
CPU code	An address and byte list of executable instructions from the Code pane of the CPU window
CPU data	An address and byte list of data in memory from the Data pane of the CPU window or the Dump window
CPU stack	A source position and stack frame from the Stack pane of the CPU window
CPU register	A register name and value from the Register pane of the CPU window or the Registers window
CPU flag	A CPU flag value from the Flags pane of the CPU window
Inspector	One of the following: ■ A variable name from an Inspector window ■ A constant value from an Inspector or Watches window ■ A register-based variable from an Inspector window ■ A bit field from an Inspector window
Address	An address without data or code attached
Expression	An expression from the Watches window
Coprocessor	An 80x87 numeric coprocessor register
Control flag	An 80x87 control flag value
Status flag	An 80x87 status flag value

The Clipboard window local menu

```
Inspect
Remove
Delete all
Freeze
```

If you're in the Clipboard window and you press *Alt-F10* or click the right mouse button, you see the menu at the left. Alternatively, you can press *Ctrl* and the highlighted key of the local menu command to execute a command.

Table 2.4
Clipboard local menu commands

Command	**Description**
Inspect	Positions the cursor in the window from which the item was clipped.
Remove	Removes the highlighted item or items. Pressing *Del* has the same effect on a highlighted item.
Delete All	Deletes everything in the Clipboard.
Freeze	Stops the Clipboard item from being dynamically updated.

Dynamic updating

The Clipboard dynamically updates any item with an associated value, such as an expression from the Watches window, a variable from the Inspector window, or a register from the CPU window. You can use the Clipboard as a large Watches window if you wish, and you can freeze the value of any item you like.

See Chapter 6 for more information on watching expressions.

For example, you might want to put a Watches window expression in the Clipboard. To do so, first put it in the Watches window, then press *Shift-F3* to copy it into the Clipboard. The value of the item then changes just as it would in a Watches window, unless you use the local menu Freeze command to disable the watchpoint.

One advantage of watching an expression in the Clipboard is that you can freeze the expression at a certain value, then continue running the program and compare the frozen value to the changing values in the Watches window.

Tips for using the Clipboard

The possible uses of the Clipboard are too numerous to list here. Some of the things you can do with it are

- clipping from lines in Module windows as a way of marking locations that you can later return to using the local menu Goto command (by pasting a location into the dialog box displayed by the Goto command)
- watching an expression (see the previous section)

- pasting new values into variables using the Data | Evaluate dialog box or the dialog box for the Change command of the Inspector window or the Watches window
- pasting strings into the Log window to help you keep track of what you did during a debugging session
- pasting an address (the *location* category of an item) into any of the places where an address is requested (such as the Breakpoints | Options dialog box Address field, or the Run | Execute To dialog box)
- pasting expressions into conditions and actions of breakpoints
- pasting parameters into the Run | Arguments dialog box
- pasting a window procedure name or an ObjectWindows object name into the Windows Messages window
- pasting a string into the dialog box for the Module window Search command
- copying data from and pasting it to the CPU data pane
- copying code from one part of the CPU window to another and then running the program with the copied code

Getting help

As you've seen, TDW goes out of its way to make debugging easy for you. It doesn't require you to remember obscure commands; it keeps lists of what you type, in case you want to repeat it; it lets you define macros; and it offers sophisticated control of your windows. To avoid potential confusion, TDW offers the following help features:

READY

- An activity indicator in the upper right corner always displays the current activity. For example, if your cursor is in a window, the activity indicator reads `READY`; if there's a menu visible, it reads `MENU`; if you're in a dialog box, it reads `PROMPT`. If you ever get confused about what's happening in TDW, look at the activity indicator for help. (Other activity indicator modes are `SIZE/MOVE`, `MOVE`, `ERROR`, `RECORDING`, `WAIT`, `RUNNING`, `HELP`, `STATUS`, and `PLAYBACK`.)
- The active window is always topmost and has a double line around it.

F1

- You can access an extensive context-sensitive help system by pressing *F1*. Press *F1* again to bring up an index of help topics from which you can select what you need.

- The status line at the bottom of the screen always offers a quick reference summary of keystroke commands. The line changes as the context changes and as you press *Alt* or *Ctrl*. Whenever you are in the menu system, the status line offers a one-line synopsis of the current menu command.

For more information on the last two avenues for help, read the following two sections.

Online help

TDW, like other Borland products, gives context-sensitive onscreen help at the touch of a single key. Help is available anytime you're in a menu or window, or when an error message or prompt is displayed.

Press *F1* to bring up a Help screen showing information pertinent to the current context (window or menu). If you have a mouse, you can also bring up help by clicking F1 on the status line. Some Help screens contain highlighted keywords that let you get additional help on that topic. Use the arrow keys to move to any keyword and then press *Enter* to get to its screen. Use the *Home* and *End* keys to go to the first and last keywords on the screen, respectively.

```
Index            Shift-F1
Previous topic     Alt-F1
Help on help
```

You can also access the onscreen help feature by choosing Help from the menu bar (*Alt-H*).

If you want to return to a previous Help screen, press *Alt-F1* or choose Previous Topic from the Help menu. From within the Help system, use *PgUp* to scroll back through up to 20 linked help screens. (*PgDn* only works when you're in a group of related screens.) To access the Help Index, press *Shift-F1* (or *F1* from within the Help system), or choose Index from the Help menu. To get help on Help, choose Help | Help on Help. To exit from Help, press *Esc*.

The status line

Whenever you're in TDW, a quick-reference help line appears at the bottom of the screen. This status line provides at-a-glance keystroke or menu command help for your current context.

In a window

The normal status line shows the commands performed by the function keys and looks like this:

Figure 2.7
The normal status line

```
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

If you hold down the *Alt* key for a second or two, the commands performed by the *Alt* keys are displayed.

Figure 2.8
The status line with *Alt* pressed

```
Alt: F2-Bkpt at F3-Close F4-Back F5-User F6-Undo F7-Instr F8-Rtn F9-To F10-Local
```

If you hold down the *Ctrl* key for a second or two, the commands performed by the *Ctrl* letter keys are displayed. This status line changes depending on the current window and current pane, and it shows the single-keystroke equivalents for the current local menu. If there are more local menu commands than can be described on the status line, only the first keys are shown. You can view all the available commands on a local menu by pressing *Alt-F10* to pop up the entire menu.

Figure 2.9
The status line with *Ctrl* pressed

```
Ctrl: I-Inspect W-Watch M-Module F-File P-Previous L-Line S-Search N-Next
```

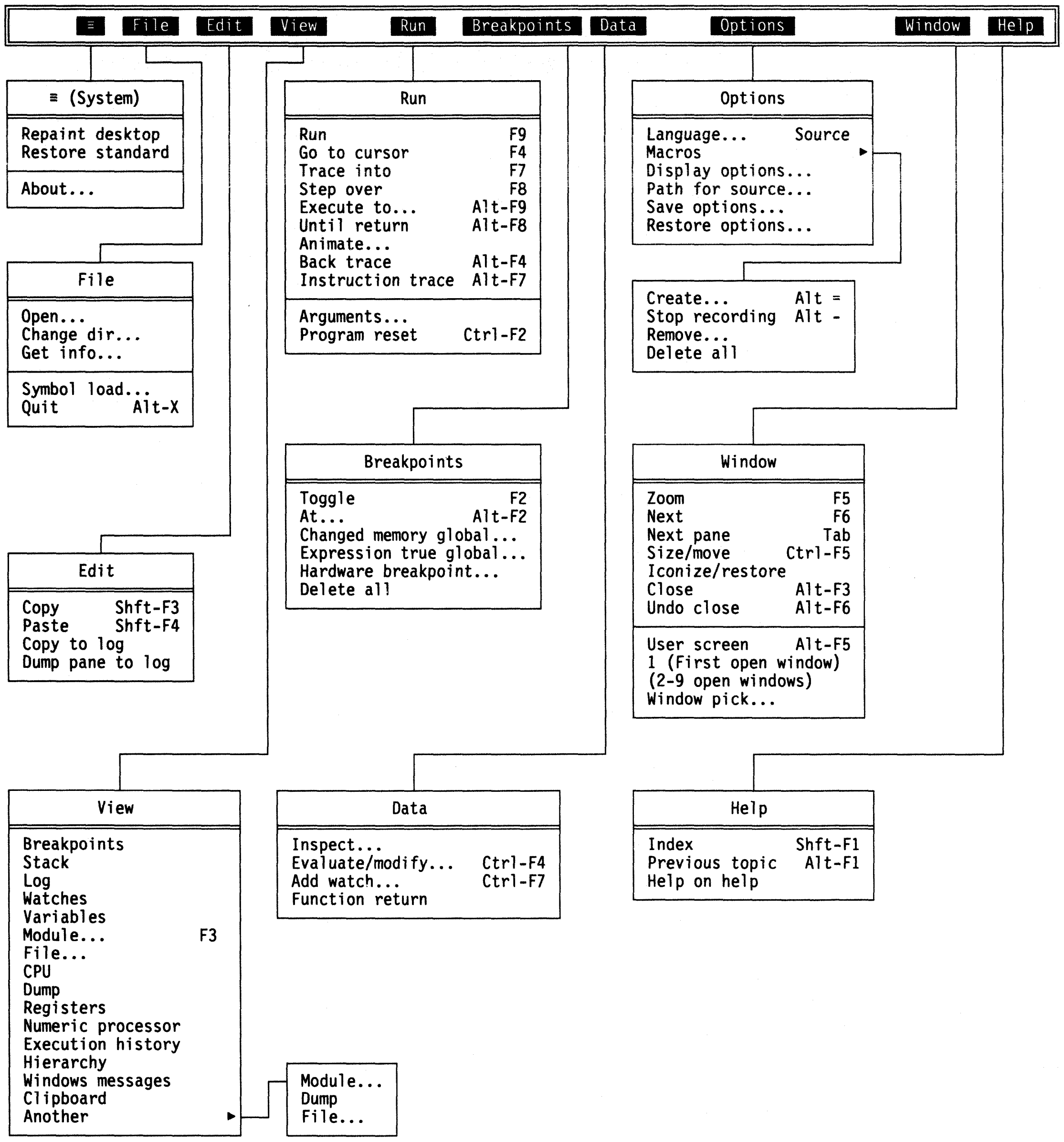
In a menu or dialog box

Whenever you are in a menu or a dialog box, the status line displays a one-line explanation of what the current item does. For example, if you have highlighted View | Registers, the status line says `Open a CPU registers window.`

The status line gives you menu help whether you are in a global menu or a local menu.

Complete menu tree

Figure 2.10: The complete Turbo Debugger menu tree



C H A P T E R

3

A quick example

If you're eager to use TDW and aren't the sort of person to work through the whole manual first, this chapter gives you enough knowledge to debug your first program. Once you've learned the basic concepts described here, the integrated environment and context-sensitive Help system make it easy to learn as you go along.

This chapter leads you through all TDW's basic features. After describing the demo program TDDEMOW provided on the distribution disks, it shows you how to

- run and stop program execution
- examine the contents of program variables
- look at complex data objects, like arrays and structures
- change the value of variables

The demo program

The demo program (TDDEMOW) introduces you to the two main things you need to know to debug a program: how to stop and start your program and how to examine your program's variables and data structures. The program itself is not meant to be particularly useful: Some of its code and data structures exist solely to show you TDW's capabilities.

The program uses the *WinCrt* unit to display its output in a window under Windows. It's not a full-featured Windows application, but it does illustrate some useful TDW concepts.

The demo program lets you type in some lines of text, then counts the number of words and letters that you entered. At the end of the program, it displays some statistics about the text, including the average number of words per line and the frequency of each letter.

Make sure your working directory contains the two files needed for the tutorial: TDDEMOW.PAS and TDDEMOW.EXE.

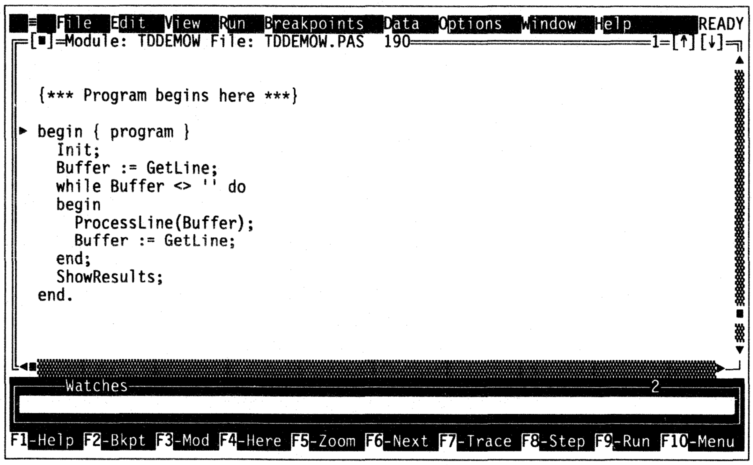
Getting in

To start the demo program,

1. Make sure Windows is running in standard or 386 enhanced mode (enhanced mode for Windows 3.1). TDW doesn't run in real mode.
2. In the Windows Program Manager, open the program group that contains Turbo Pascal and highlight the Turbo Pascal icon.
3. Choose File | Open and enter the full path to TDDEMOW.PAS.
4. When TDDEMOW.PAS comes up in the Edit window, choose Run | Debugger to run TDW and load the demo. Turbo Pascal compiles the program with debugging information for you if necessary.

TDW loads the demo program, displays the startup screen, and positions the cursor at the start of the program.

Figure 3.1
The startup screen showing TDDEMOW


≡ File Edit View Run Breakpoints Data Options Window Help READY
[■]=Module: TDDEMOW File: TDDEMOW.PAS 190 1=[↑][↓]
Watches 2
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

```
{*** Program begins here ***}

▶ begin { program }
    Init;
    Buffer := GetLine;
    while Buffer <> '' do
    begin
      ProcessLine(Buffer);
      Buffer := GetLine;
    end;
    ShowResults;
  end.
```

The startup screen consists of the menu bar, the Module and Watches windows, and the status line.

Getting out To exit from TDW at any time, press *Alt-X*. If you get hopelessly lost following the tutorial, press *Ctrl-F2* to reload the program and start at the beginning. However, *Ctrl-F2* doesn't clear breakpoints or watches; you'll have to use *Alt-B D* to do that.

Getting Help Press *F1* whenever you need Help with the current window, menu command, dialog box, or error message. You can learn a lot by working your way through the menu system and pressing *F1* at each command to get a summary of what it does.

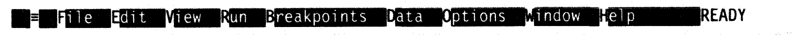
F1

Using TDW

This section discusses how to use TDW's menus and windows and the status line that appears below a window.

The menus

The top line of the screen shows the menu bar. To pull down a menu from it, press *F10*. Then, to choose a menu command, you can either use ← or → to highlight your selection and press *Enter*, or press *Alt* in combination with the highlighted letter of one of the menu names.



```
≡ File Edit View Run Breakpoints Data Options Window Help        READY
```

Figure 3.2
The menu bar

F10

Press *F10* now. Notice that the cursor disappears from the Module window, and the ≡ command on the menu bar becomes highlighted. The bottom line of the screen also changes to indicate what sort of functions the ≡ menu performs.

Use the arrow keys to move around the menu system. Press ↓ to pull down the menu for the highlighted item on the menu bar.

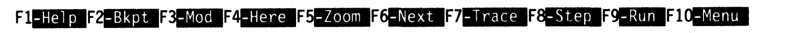
You can also open a menu by clicking an item in the menu bar with your mouse.

Esc

Press *Esc* to move back through the levels of the menu system. When just one menu item on the menu bar is highlighted, pressing *Esc* returns you to the Module window, with the menu bar no longer active.

The status line

The status line at the bottom of the screen shows relevant function keys and what they do.



```
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Figure 3.3
The status line

This line changes depending on what you are entering (menu commands, data in a dialog box, and so on). Hold *Alt* down for a second or two, for example. Notice that the status line changes to show you the function keys you can use with *Alt*.

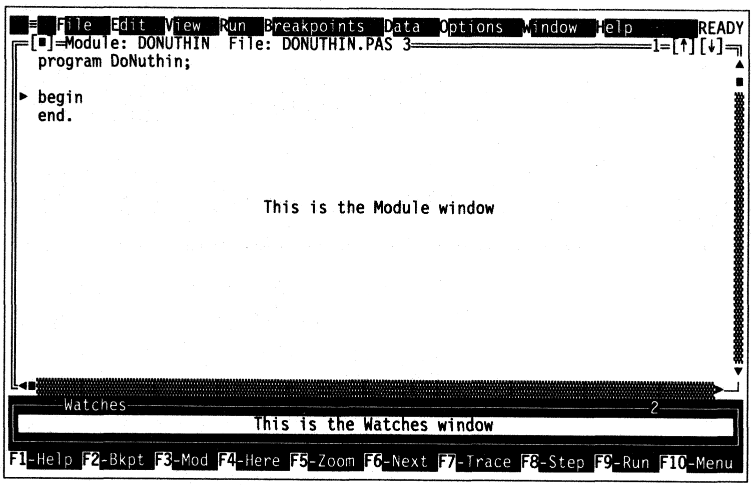
Now press *Ctrl* for a second. The commands shown on the status line are the hot keys to the *local menu commands* for the current *pane* (area of the window). They change depending on which sort of window and which pane you are in. (More about these later.)

As soon as you enter the menu system, the status line changes again to show you what the currently highlighted menu option does. Press *F10* to go to the menu bar, and press → to highlight the File option. The status line now reads, `File oriented functions.` Use ↓ to scroll through the options on the File menu, and watch the message change. Press *Esc* or click the Module window with your mouse to leave the menu system.

The windows

The window area takes up most of the screen. This area is where you examine various parts of your program through the different windows.

The display starts up with two windows: a Module window and the Watches window. Until you open more windows or adjust these two, they remain *tiled*, filling the entire screen without overlapping. New windows automatically overlap existing windows until you move them.



≡ File Edit View Run Breakpoints Data Options Window Help READY
[■]=Module: DONUTHIN File: DONUTHIN.PAS 3 1=[↑][↓]
Watches 2
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

```
program DoNuthin;

► begin
  end.
```

This is the Module window

This is the Watches window

Figure 3.4
The Module and Watches windows, tiled

F6

Notice that the Module window has a double-line border and a highlighted title, which indicate that it's the active window. You use the cursor keys (the arrow keys, *Home, End, PgUp,* and so on) to move around inside the active window. Now press *F6* to switch to another window. The Watches window becomes active, with a double-line border and a highlighted title.

You use commands from the View menu to create new windows. For example, choose View | Stack to open a Stack window. The Stack window pops up on top of the Module window.

Alt F3

Now press *Alt-F3* to remove the active window. The Stack window disappears.

Alt F6

TDW stores the last-closed window, making it possible for you to recover it if you need to. If you accidentally close a window, choose Window | Undo Close. If you do so now, you see the Stack window reappear. You can also press *Alt-F6* to recover the last-closed window.

The Window menu contains the commands that let you adjust the appearance of the windows you already have onscreen. You can both move the window around the screen and change its size. (You can use *Ctrl-F5* to do the same thing.)

Choose Window | Size/Move and use the arrow keys to reposition the active window (the Stack window) on the screen. Next, hold *Shift* down and use the arrow keys to adjust the size of the window. Press *Enter* when you have defined a new size and position that you like.

Now, to prepare for the next section, remove the Stack window by pressing *Alt-F3*. Then continue with the next section.

Using the TDDEMOW sample program

F7

The filled arrow (►) in the left column of the Module window shows where TDW stopped your program. Since you haven't run your program yet, the arrow is on the first line of the program. Press *F7* to trace a single source line. The arrow and cursor are now on the next executable line.

Look at the right margin of the Module window title. It shows the line that the cursor is on. Move the cursor up and down with the arrow keys and notice how the line number in the title changes.

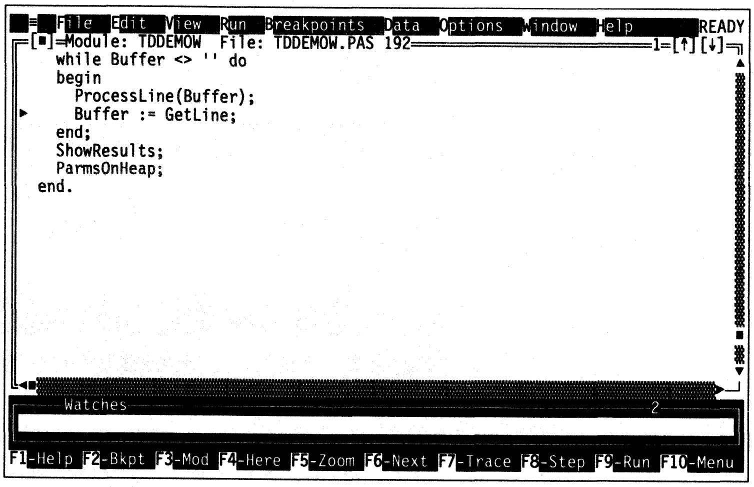
F4

To make the program execute until it reaches line 189, move the cursor to that line and then press *F4*. TDDEMOW prompts you to enter a string. Type `ABC`, a space, `DEF`, and then press *Enter*. Now, with the cursor still on line 189, press *F7* twice to execute two more lines of source code. Since the second line you executed is a call to a different procedure, the arrow now appears on the first line of the function *ProcessLine*. Continuing to press *F7* would step you through the function *ProcessLine* and then return you to the line following the call—line 192. Instead, press *Alt-F8* to make the program execute *ProcessLine* and then stop when *ProcessLine* returns. This command is very useful when you want to jump past the end of a function or procedure.

Alt F8

F8

If you had pressed *F8* instead of *F7* on line 189, the cursor would have gone directly to line 192 instead of into the function. *F8* is similar to *F7* in that it executes procedures, but it doesn't step through their source code.



File Edit View Run Breakpoints Data Options Window Help READY
[■]=Module: TDDEMOW File: TDDEMOW.PAS 192 1=[↑][↓]
Watches 2
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

```
  while Buffer <> '' do
  begin
    ProcessLine(Buffer);
►   Buffer := GetLine;
  end;
  ShowResults;
  ParmsOnHeap;
end.
```

Figure 3.5
The program stops after returning from a procedure

Alt F9

To execute the program until it reaches a specific place, you can name the function or line number directly, instead of moving the cursor to that line in a source file and then running to that point. Press *Alt-F9* to specify a label to run to. A dialog box appears. Type `GetLine` and press *Enter*. The program runs, then stops at the beginning of function *GetLine*.

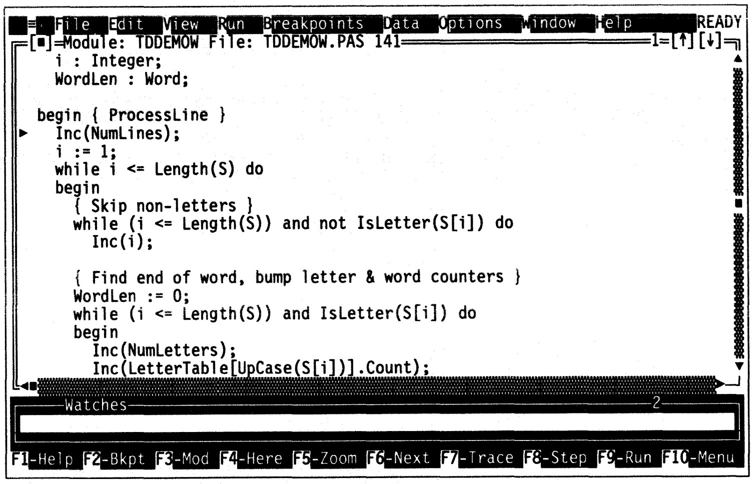
Setting breakpoints

F2

Another way to control where your program stops running is to set breakpoints. The simplest way to set a breakpoint is with the *F2* key. Move the cursor to line 141 and press *F2*. TDW highlights the line, indicating there is a breakpoint set on it. (A quick way to get to line 141 is to press *Ctrl-L*, the hot key for the Line command in the Module window local menu, then type `141` in the text entry box and press *Enter*.)

You can also use the mouse to toggle breakpoints by clicking the first two columns of the Module window.

Figure 3.6
A breakpoint at line 141


≡ File Edit View Run Breakpoints Data Options Window Help READY
[■]=Module: TDDEMOW File: TDDEMOW.PAS 141 1=[↑][↓]

```
  i : Integer;
  WordLen : Word;

begin { ProcessLine }
► Inc(NumLines);
  i := 1;
  while i <= Length(S) do
  begin
    { Skip non-letters }
    while (i <= Length(S)) and not IsLetter(S[i]) do
      Inc(i);

    { Find end of word, bump letter & word counters }
    WordLen := 0;
    while (i <= Length(S)) and IsLetter(S[i]) do
    begin
      Inc(NumLetters);
      Inc(LetterTable[UpCase(S[i])].Count);
```

Watches 2
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

F9

Now press *F9* to execute your program without interruption. The screen switches to the program's display. The demo program is now running and waiting for you to enter a line of text. Type `abc`, a space, `def`, and then press *Enter*. The display returns to the TDW screen with the arrow on line 141, where you set a breakpoint that has stopped the program. Now press *F2* again to toggle it off.

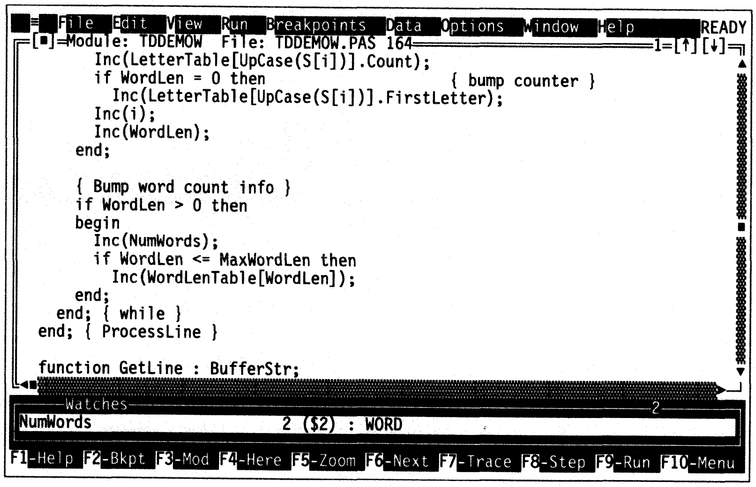
See Chapter 7 for a complete description of breakpoints, including conditional and global breakpoints.

Using watches

Alt F10

The Watches window at the bottom of the screen shows the value of variables you specify. For example, to watch the value of the variable *NumWords*, move the cursor to the variable name on line 164. (A quick way to get there is to press *Ctrl-S*, the hot key for the Search command in the Module window local menu, then type `NumWords` in the text entry box and press *Enter*.) Then choose Watch from the Module window local menu (bring it up with *Alt-F10* or the right mouse button, or use the shortcut *Ctrl-W*).

Figure 3.7
A Pascal variable in the Watches window


≡ File Edit View Run Breakpoints Data Options Window Help READY
[■]=Module: TDDEMOW File: TDDEMOW.PAS 164 1=[↑][↓]
Watches 2
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

```
      Inc(LetterTable[UpCase(S[i])].Count);
      if WordLen = 0 then                  { bump counter }
        Inc(LetterTable[UpCase(S[i])].FirstLetter);
      Inc(i);
      Inc(WordLen);
    end;

    { Bump word count info }
    if WordLen > 0 then
    begin
      Inc(NumWords);
      if WordLen <= MaxWordLen then
        Inc(WordLenTable[WordLen]);
    end;
  end; { while }
end; { ProcessLine }

function GetLine : BufferStr;
```

```
NumWords                        2 ($2) : WORD
```

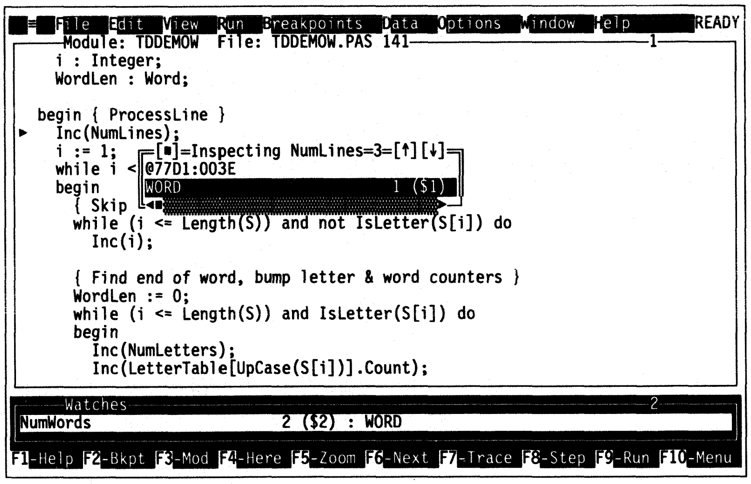
NumWords now appears in the Watches window at the bottom of the screen, along with its type (Word) and value. As you execute the program, TDW updates this value to reflect the variable's current value.

Examining simple Pascal data objects

Once you have stopped your program, there are a number of ways of looking at data using the Inspect command. This very powerful facility lets you examine data structures in the same way that you visualize them when you write a program.

The Inspect commands (in various local menus and in the Data menu) let you examine any variable you specify. Suppose you want to look at the value of the variable *NumLines*. Move the cursor back to line 141 so it's under one of the letters in *NumLines* and press *Ctrl-I*. An Inspector window pops up.

Figure 3.8
An Inspector window


≡ File Edit View Run Breakpoints Data Options Window Help READY
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

```
Module: TDDEMOW  File: TDDEMOW.PAS 141                                1
   i : Integer;
   WordLen : Word;

 begin { ProcessLine }
►  Inc(NumLines);
   i := 1;   [■]=Inspecting NumLines=3=[↑][↓]
   while i < @77D1:003E
   begin     WORD                            1 ($1)
     { Skip
     while (i <= Length(S)) and not IsLetter(S[i]) do
       Inc(i);

     { Find end of word, bump letter & word counters }
     WordLen := 0;
     while (i <= Length(S)) and IsLetter(S[i]) do
     begin
       Inc(NumLetters);
       Inc(LetterTable[UpCase(S[i])].Count);

Watches                                                              2
NumWords                        2 ($2) : WORD
```

The first line tells you the variable name; the second line shows its address in memory. The third line tells you what type of data is stored in *NumLines* (it's a Pascal Word) and displays the current value of the variable.

Now, having examined the variable, press *Esc* to close the Inspector window. You can also use *Alt-F3* to remove the Inspector window, just like any other window, or you can click the close box with your mouse.

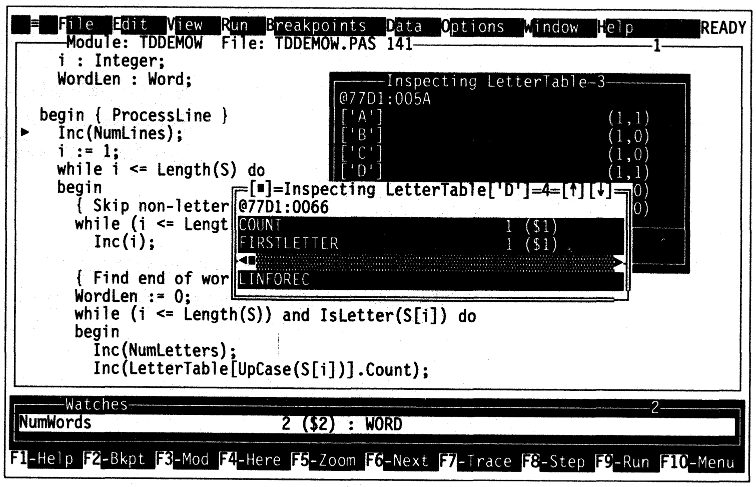
Let's review what you actually did here. By pressing *Ctrl*, you used a hot key for the local menu commands in the Module window. Pressing *I* specified the Inspect command.

To examine a data item that is not conveniently displayed in the Module window, choose Data | Inspect. A dialog box appears, asking you to enter the variable to inspect. Type `LetterTable` and press *Enter*. An Inspector window appears, showing the value of *LetterTable*. Use the arrow keys to scroll through the 26 elements that make up *LetterTable*. The title of the Inspector window shows the name of the data you are inspecting. The next section shows you how to examine this compound data object.

Examining compound Pascal data objects

A compound data object, such as an array or structure, contains multiple components. Move to the fourth element of the *LetterTable* array (the one indicated by `['D']`). Press *Alt-F10* to bring up the local menu for the Inspector window, then choose Inspect. A new Inspector window appears, showing the contents of that element in the array. This Inspector window shows the contents of a record of type *LInfoRec*.

Figure 3.9
Inspecting a record


≡ File Edit View Run Breakpoints Data Options Window Help READY
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

```
Module: TDDEMOW  File: TDDEMOW.PAS 141                                    1
   i : Integer;
   WordLen : Word;

 begin { ProcessLine }
▶  Inc(NumLines);
   i := 1;
   while i <= Length(S) do
   begin
     { Skip non-letter
     while (i <= Lengt
       Inc(i);

     { Find end of wor
     WordLen := 0;
     while (i <= Length(S)) and IsLetter(S[i]) do
     begin
       Inc(NumLetters);
       Inc(LetterTable[UpCase(S[i])].Count);
```

```
Inspecting LetterTable-3
@77D1:005A
['A']                    (1,1)
['B']                    (1,0)
['C']                    (1,0)
['D']                    (1,1)
```

```
[■]=Inspecting LetterTable['D']=4=[↑][↓]
@77D1:0066
COUNT                    1 ($1)
FIRSTLETTER              1 ($1)
LINFOREC
```

```
Watches                                                                   2
NumWords                  2 ($2) : WORD
```

When you place the cursor over one of the member names, the data type of that member appears in the bottom pane of the Inspector window. If one of these members were in turn a compound data object, you could issue an Inspect command and dig down further into the data structure.

Alt F3

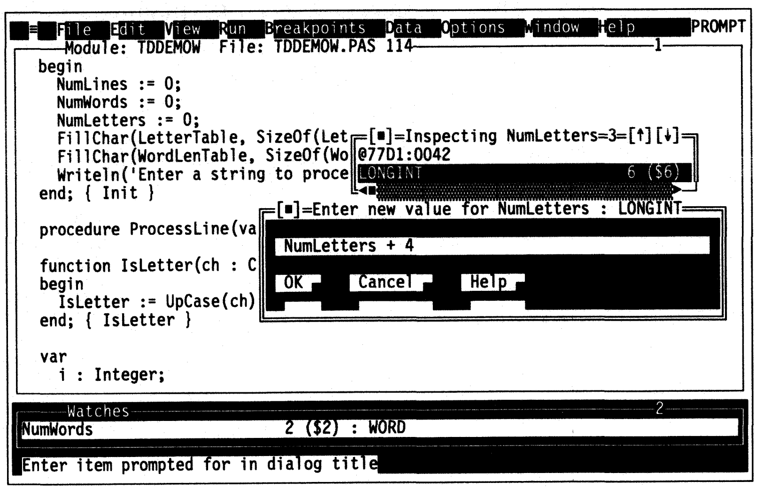
Press *Alt-F3* to remove both Inspector windows and return to the Module window. (*Alt-F3* is a convenient way of removing several Inspector windows at once. If you had pressed *Esc*, only the topmost Inspector window would have been deleted.)

Changing Pascal data values

So far, you've learned how to *look* at data in the program. Now, let's *change* the value of data items.

Use the arrow keys to go to line 114 in the source file. Place the cursor at the variable called *NumLetters* and press *Ctrl-I* to inspect its value. In the Inspector window, press *Alt-F10* to bring up the Inspector window's local menu. Choose the Change option. (You could also have done this directly from the Inspector window by pressing *Ctrl-C*.) A dialog box appears, asking for the new value.

Figure 3.10
The Change dialog box


≡ File Edit View Run Breakpoints Data Options Window Help PROMPT
Module: TDDEMOW File: TDDEMOW.PAS 114
1
[■]=Inspecting NumLetters=3=[↑][↓]
[■]=Enter new value for NumLetters : LONGINT
OK
Cancel
Help
Watches
2

```
begin
  NumLines := 0;
  NumWords := 0;
  NumLetters := 0;
  FillChar(LetterTable, SizeOf(Let
  FillChar(WordLenTable, SizeOf(Wo
  Writeln('Enter a string to proce
end; { Init }

procedure ProcessLine(va

function IsLetter(ch : C
begin
  IsLetter := UpCase(ch)
end; { IsLetter }

var
  i : Integer;
```

```
@77D1:0042
LONGINT                    6 ($6)
```

```
NumLetters + 4
```

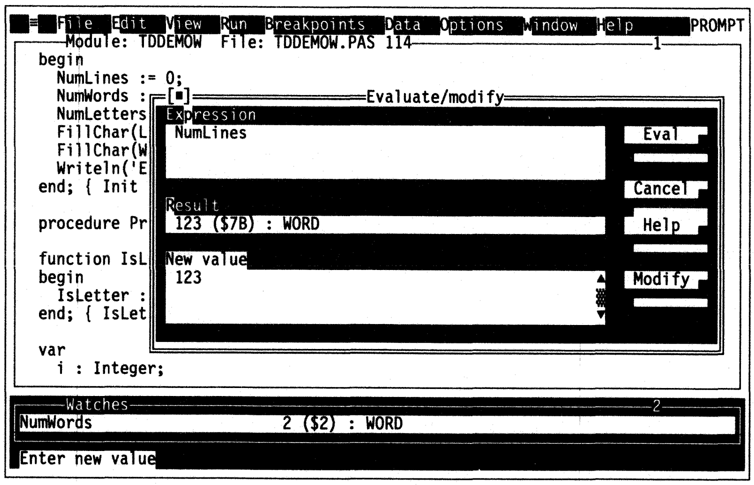
```
NumWords                   2 ($2) : WORD
```

```
Enter item prompted for in dialog title
```

At this point, you can enter any Pascal expression that evaluates to a number. Type `NumLetters + 4` and press *Enter*. The value in the Inspector window now shows the new value, `10 ($A)`.

To change a data item that isn't displayed in the Module window, choose Data | Evaluate/Modify. A dialog box appears. Enter the name of the variable to change. Type `NumLines` and press *Enter*. The result is displayed in the middle pane. Press *Tab* twice, then type `123` and press *Enter*. This sets the variable *NumLines* to *123*.

Figure 3.11
The Evaluate/Modify dialog box


```
≡ File Edit View Run Breakpoints Data Options Window Help      PROMPT
┌──Module: TDDEMOW  File: TDDEMOW.PAS 114──────────────────1──┐
  begin
    NumLines := 0;
    NumWords :╔[■]═══════════Evaluate/modify═══════════╗
    NumLetters║ Expression
    FillChar(L║  NumLines                        Eval
    FillChar(W║
    Writeln('E║
  end; { Init ║                                  Cancel
              ║ Result
  procedure Pr║  123 ($7B) : WORD                Help

  function IsL║ New value
  begin       ║  123                             Modify
    IsLetter :║
  end; { IsLet║
              ╚═══════════════════════════════════════╝
  var
    i : Integer;
┌──Watches─────────────────────────────────────────────2──┐
 NumWords                    2 ($2) : WORD
Enter new value
```

That wraps up our quick introduction to using TDW with a Turbo Pascal for Windows program. Chapter 13 offers a more extensive debugging sample.

C H A P T E R

4

Starting TDW

This chapter tells you how to prepare programs for debugging. We show you how to start TDW from Windows and how to tailor its many command-line options to suit the program you are debugging. We explain how to make these options permanent in a configuration file and, finally, how to return to Windows when you are done.

Preparing programs for debugging

When you compile and link with Turbo Pascal for Windows, you can tell the compiler to generate full debugging information by checking Options | Linker | Debug Info in EXE and Options | Compiler | Debug Information.

If you have compiled your program's modules without any debugging information and you select Run | Debugger from a Turbo Pascal edit window, Turbo Pascal will recompile your program with debug information before launching TDW.

If you need to recompile your modules with debugging information, it's possible to generate debug information only for specific modules (you might have to do this if you're debugging a large program), but you will find it annoying later to enter a module that doesn't have any debug information available. We suggest recompiling all modules.

If you're using the integrated environment of Turbo Pascal for Windows, you need to do the following before compiling to put debug information into your .EXE or DLL file:

- Use the Options | Linker command to bring up the Linker Options window, then check the Debug Info in EXE check box.
- Use the Options | Compiler command to bring up the Compiler Options window, then check the Debug Information check box. Alternatively, you can use the {**$D+**} compiler directive.
- If you want to be able to access local symbols (any declared within procedures and functions), you must either use Options | Compiler to bring up the Compiler Options window and then check the Local Symbols check box, or put the following directive at the start of your program:

Enter the directive just like this, with no spaces.

```
{$L+}
```

If you're using the command-line compiler (TPCW.EXE), you must compile using the **/v** command-line option. Debug information and local symbols are, by default, generated. If you don't want them, use the command-line options **/$D-** and **/$L-** to disable them.

Starting TDW

There are four ways to run TDW:

- If you are in Turbo Pascal for Windows, you can debug the program in the active window by choosing Run | Debugger.
- If you are in Windows, the easiest method is to open the appropriate program group in the Windows Program Manager and choose the TDW icon. Then choose File | Open to load the program you're debugging.

  Warning!

  For this and the next option, unless TDW is in your path and your program is in your Windows directory, you must be careful to type in the correct path for both TDW and your application.
- If you are in Windows and you want to enter command-line options, you can start TDW by using the Windows Program Manager File | Run command to open the Run dialog box. Then, in the Command Line input box, just type `TDW`, followed by any command-line options and, optionally, the name of the program you're debugging, as if you were at the DOS prompt.

- If you are at the DOS prompt, you can start TDW by entering the following and pressing *Enter*:

```
WIN TDW [options] [progname [progargs]]
```

Entering command-line options

If you start TDW from the DOS prompt or by using the Program Manager File | Run command, you can add command-line options after typing `TDW`.

If you start TDW from Turbo Pascal for Windows, you can enter command-line options by using Run | Parameters.

You can also indicate options in the TPW.INI file.

Directly entering command-line options

The generic command-line format is

```
TDW [options] [progname [progargs]]
```

The items enclosed in brackets are optional; if you include any, type them without the brackets. *Progname* is the name of the program to debug.

You can follow a program name with arguments. Here are some sample command lines:

Command	Action
`tdw -tc:\prog1 prog1 a b`	Starts the debugger in the C:\PROG1 directory and loads program *prog1* with two command-line arguments, *a* and *b*.
`tdw prog2 -x`	Starts the debugger with default options and loads program *prog2* with one argument, *-x*.

If you simply type `TDW` *Enter*, TDW loads and uses its default options.

Indicating command-line options in TPW.INI

If you start TDW from Turbo Pascal and you want to indicate command-line options or change the path to TDW.EXE, you have to edit the Turbo Pascal for Windows initialization file, TPW.INI. This file is located in the same directory as the Windows program (WINDOWS is the default name).

When you edit TPW.INI, you see a Startup section, followed by one or more Startup settings. For example,

```
[Startup]
CfgPath=C:\TPW\TPW.CFG
SizeOrg=2,44,44,596,377
```

You can add TDW information to the file after the startup information. To indicate that the information is for TDW, you must start a new section called Debugger. This section can contain two settings, Exepath and Switches. The format of the Debugger section is as follows:

```
[Debugger]
Exepath=<pathname>
Switches=<command-line options>
```

Exepath	Enter the path to TDW.EXE. This setting is necessary if you have moved TDW.EXE somewhere besides the directory where the INSTALL program put it.
Switches	Enter one or more TDW command-line options separated by spaces. Do not enter the name of the program you want TDW to load; Turbo Pascal determines it for you.

For example, if you've moved TDW.EXE to a directory on your C drive called TDW and you want to start TDW in assembler mode and set the source directory to C:\MYAPP\SOURCE, your Debugger section would look like the following:

```
[Debugger]
Exepath=C:\TDW
Switches=-l -sdc:\myapp\source
```

Things to remember

When you run a program in TDW, you need to have *both* its .EXE and .DLL files and the original source files available. TDW searches for source files first in the directory the compiler found them in when it compiled, second in the directory specified in the Options | Path for Source command, third in the current directory, and fourth in the directory the .EXE or .DLL file is in.

☞ You must already have compiled your source code into an executable (.EXE or .DLL) file with full debugging information turned on before debugging with TDW.

TDW works only with Windows programs compiled with a Borland compiler.

If you're running your program from Windows and notice a bug, you have to exit your program and load it under TDW before you can begin debugging.

All .EXE and .DLL files for the application must be in the same directory.

Running TDW

When you run TDW, it comes up in full-screen character mode, not in a window. Despite this appearance, TDW is a Windows application and will run only under Windows.

Unlike other applications that run under Windows, you can't use the Windows shortcut keys (like *Alt-Esc* or *Ctrl-Esc*) to switch out of the TDW display and run another program. However, if the application you are debugging is active (the cursor is active in one of its windows), you can use *Alt-Esc, Ctrl-Esc,* or the mouse to switch to other programs.

If you do use *Ctrl-Esc* to switch out of an application running under TDW, you see the application name on the list of tasks. You will never see TDW on the task list because TDW is not a normal Windows task that you can switch into or out of.

Command-line options

The end of this chapter has a complete list of TDW's command-line options.

All TDW command-line options start with a hyphen (-) and are separated from the TDW command and each other by at least one space. You can explicitly turn a command-line option off by following the option with another hyphen. For example, **-p-** disables the mouse. Turning a command-line option off works even if an option has been permanently enabled in the configuration file. You can modify the configuration file by using the TDWINST configuration program described in the online text file UTILS.TDW.

The following sections describe all available TDW command-line options.

Loading the configuration file (-c)

This option loads the specified configuration file. There must not be a space between **–c** and the file name.

If the **–c** option isn't included, TDCONFIG.TDW is loaded if it exists. Here's an example:

```
TDW -cMYCFG.TDW TDDEMOW
```

This command loads the configuration file MYCONF.TDW and the source code for TDDEMOW.

Display updating (-d)

The **–d** options affect the way in which display updating is performed.

–do	Runs TDW on your secondary display. View your program's screen on the primary display, and run the debugger on the secondary one.
–ds	The default option for all displays, it's also called *screen swapping*. Required if your only display is monochrome. Maintains a separate screen image for the debugger and the program being debugged by loading the entire screen from memory each time your program is run or the debugger is restarted. This technique is the most time-consuming method of displaying the two screen images, but works on any display hardware and with programs that do unusual things to the display.

Getting help (-h and –?)

These options display a window that describes TDW's command-line syntax and options.

Assembler-mode startup (-l)

This option forces startup in assembler mode, showing the CPU window. TDW does not execute your program's startup code, which usually executes automatically when you load your program into the debugger. This means that you can step through your startup code.

If you are debugging a DLL, this option also allows you to debug the assembly-language code that starts up the DLL. See Chapter 11, page 173, for more information on debugging DLLs.

Mouse support (-p)

This option enables mouse support. However, since the default for mouse support in TDW is *On*, you won't have much use for the **–p** option unless you use TDWINST to change the default to *Off*. If you want to disable the mouse, use **–p–**.

☞ If the mouse driver is disabled for Windows, it will be disabled for TDW as well, and the **–p** command-line option will have no effect.

Source code handling (-s)

–sc Ignores case when you enter symbol names, even if your program has been linked with case sensitivity enabled.

Without the **–sc** option, Turbo Debugger ignores case only if you've linked your program with the case ignore option enabled.

This option doesn't change the starting directory.

–sd Sets one or more source directories to scan for source files; the syntax is

```
-sddirname[;dirname...]
```

To set multiple directories, use multiple *dirnames* separated with semicolons (;) with the **–sd** option or use the **–sd** option repeatedly or both. TDW searches for directories in the order specified. *dirname* can be a relative or absolute path and can include a disk letter. If the configuration file specifies any directories, the ones specified by the **–sd** option are added to the end of that list.

Starting directory (-t)

This option changes TDW's starting directory, which is where TDW looks for the configuration file and for .EXE files not specified with a full path. There must not be a space between the option and the directory path name.

–t<dir> Set the starting directory to <dir>. The syntax is

```
-tdirname
```

You can set only one starting directory with this option. If you enter multiple directories for one **–t** option, TDW ignores all the directories. If you enter the option more than once on the same command line, TDW uses only the last entry.

For example, the following entry would start TDW in the D:\WORKING directory:

```
tdw -tc:\utils\screensv -td:\working
```

Configuration files

TDW has two configuration files.

- TDCONFIG.TDW is created and altered by the TDWINST program and overrides command-line options to set things like display colors and display options.
- TDW.INI, installed in the Windows system directory by the installation program, has settings for special video adapters and indicates the location of the Windows-debugging DLL TDWIN.DLL.

See page 3 and the README file for more information on TDW.INI.

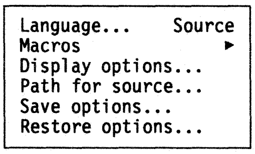
See the file UTILS.TDW for a description of how to use TDWINST to create configuration files.

TDW uses a configuration file to override built-in default values for command-line options. You can use TDWINST to set the options that TDW will default to if there is no configuration file. You can also use it to build configuration files.

TDW looks for the configuration file TDCONFIG.TDW first in the current directory, next in the TDW directory set up with the Turbo Pascal for Windows installation program, and then in the directory that contains TDW.EXE.

If TDW finds a configuration file, the settings in that file override TDW's built-in defaults. Any command-line options that you supply when you start TDW from DOS override both the corresponding default options and any corresponding values in TDCONFIG.TDW.

The Options menu

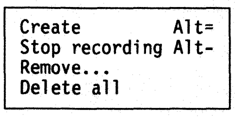


The Options menu lets you set or adjust a number of parameters that control the overall appearance and operation of TDW. The following sections describe each menu command and refer you to other sections of the manual where you can find more details.

The Language command

Chapter 9 describes how to set the current expression language and how it affects the way you enter expressions.

The Macros menu



The Macros command displays another menu that lets you define new keystroke macros or delete ones that you have already assigned to a key. It has the following commands: Create, Stop Recording, Remove, and Delete All.

Create

When issued, the Create command starts recording keystrokes into an assigned macro key. As an alternative, press the *Alt=* (Alt-Equal) hot key for Create.

When you choose Create to start recording, a prompt asks for a key to assign the macro to. Respond by typing in a keystroke or combination of keys (for example, *Shift-F9*). The message `RECORDING` will be displayed in the upper right corner of the screen while you record the macro.

Stop Recording

The Stop Recording command terminates the macro recording session. Use the *Alt-* (Alt-Hyphen) hot key to issue this command or press the macro keystroke that you are defining to stop recording.

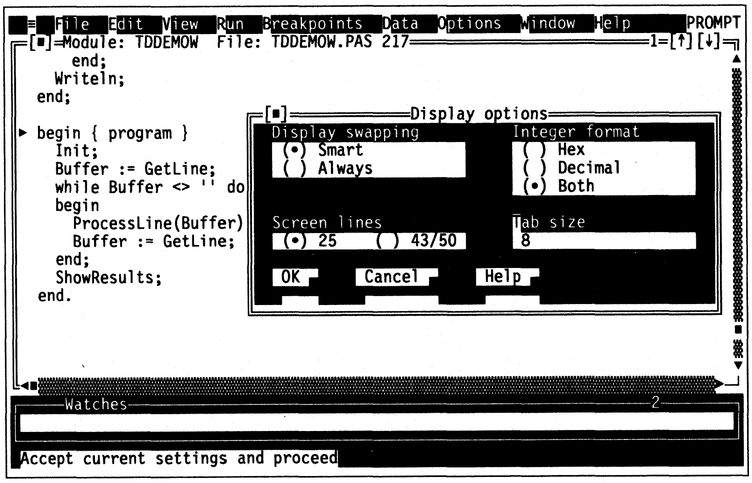
Do *not* use the Options | Macro | Stop Recording menu selection to stop recording your macro, as these keystrokes will then be added to your macro! (The menu item is added to remind you of the *Alt-* hot key.)

Remove Displays a dialog box listing all current macros. To delete a macro, select one from the list and press *Enter.*

Delete All Removes all keystroke macro definitions and restores all keys to the meaning that they originally had.

Display Options command

This command opens a dialog box in which you can set several options that control the appearance of the TDW display.



```
[■]=Module: TDDEMOW  File: TDDEMOW.PAS 217                1=[↑][↓]
      end;
    Writeln;
  end;

▶ begin { program }
    Init;
    Buffer := GetLine;
    while Buffer <> '' do
    begin
      ProcessLine(Buffer)
      Buffer := GetLine;
    end;
    ShowResults;
  end.
```

Display options

Display swapping
(•) Smart
( ) Always

Integer format
( ) Hex
( ) Decimal
(•) Both

Screen lines
(•) 25 ( ) 43/50

Tab size
8

Accept current settings and proceed

Figure 4.1
The Display Options dialog box

Display Swapping The Display Swapping radio buttons let you choose from two ways of controlling how the User screen gets swapped back and forth with TDW's screen:

Smart Swap to the User screen only when display output may occur. TDW swaps the screens any time that you step over a routine.

Always Swap to the User screen every time the user program runs. Use this option if the Smart option is not catching all the occurrences of your program writing to screen. If you choose this option, the screen flickers every time you step through your program because TDW's screen is replaced for a short time with the User screen.

Integer Format These radio buttons let you choose from three formats for displaying integers:

Hex	Shows integers as hexadecimal numbers, displayed in a format appropriate to the current language.
Decimal	Shows integers as ordinary decimal numbers.
Both	Shows integers as both decimal numbers and as hex numbers in parentheses after the decimal value.

Screen Lines These radio buttons are used to determine whether TDW's screen uses the normal 25-line display or the 43- or 50-line display available on EGA and VGA display adapters.

Tab Size This input box lets you set how many columns each tab stop occupies. You can reduce the tab column width to see more text in source files that have a lot of code indented with tabs. You can set the tab column width from 1 to 32.

Path for Source command

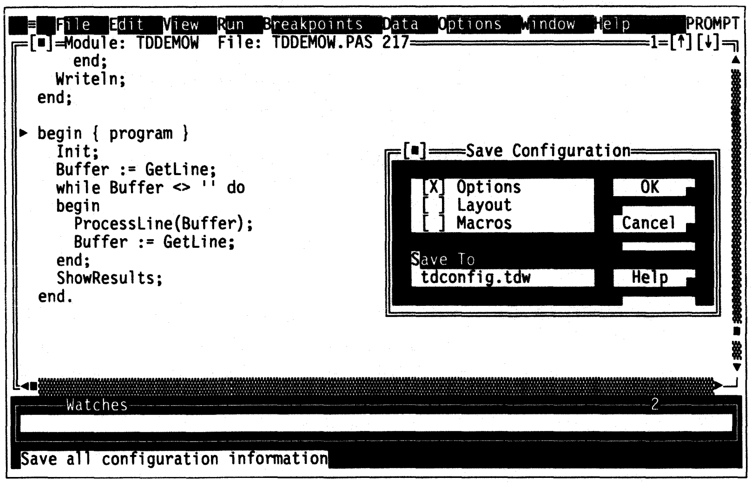
Sets the directories that TDW searches for your source files. See the discussion of the Module window in Chapter 8 for more information.

Save Options command

This command opens a dialog box from which you can save your current options to a configuration file on disk. The options you can save are

- your macros
- the current window layout and pane formats
- all settings made in the Options menu

Figure 4.2
The Save Options dialog box


File Edit View Run Breakpoints Data Options Window Help PROMPT
Save Configuration
OK
Cancel
Help
Watches

```
[■]=Module: TDDEMOW File: TDDEMOW.PAS 217
    end;
  Writeln;
end;

begin { program }
  Init;
  Buffer := GetLine;
  while Buffer <> '' do
  begin
    ProcessLine(Buffer);
    Buffer := GetLine;
  end;
  ShowResults;
end.
```

```
[X] Options
[ ] Layout
[ ] Macros
Save To
tdconfig.tdw
```

```
Save all configuration information
```

TDW lets you save your options in any or all of these ways, depending on which of the Save Configuration check boxes you turn on:

Options	Saves all settings made in the Options menu.
Layout	Saves the windowing layout.
Macros	Saves the currently defined macros.

You can also use the Save To input box to change the name of the configuration file to which you are saving the options.

Restore Options command

Restores your options from a disk file. You can have multiple configuration files, containing different macros, window layouts, and so forth. You must choose a configuration file that was created with the Save Options command or with TDWINST.

Returning to Windows

You can end your debugging session and return to the Windows Program Manager at any time by pressing *Alt-X*, except when a dialog box is active (in that case, first close the dialog box by pressing *Esc*). You can also choose File | Quit.

Summary of command-line options

When you start up TDW from the Windows Program Manager File | Run command, you can at the same time configure it using certain options. Here's the general format to use:

```
TDW [options] [program_name [program_args]]
```

Items enclosed in brackets are optional. Following an option with a hyphen disables that option if it was already enabled in the configuration file.

Table 4.1
TDW command-line options

Option	What it means
–c*filename*	Startup configuration file
–do	Other display
–ds	Swap user screen contents
–h, –?	Display help screen listing all the command-line options
–l	Assembler startup code debugging for applications and DLLs (the letter in this option is a lowercase L)
–p	Enable mouse
–sc	No case-checking of symbols
–sd*dir*[;*dir*...]	Source file directory
–t*directory*	Set starting directory for loading configuration and executable files

C H A P T E R

5

Controlling program execution

When you debug a program, you usually execute portions of it and check at a stopping point to see that it is behaving correctly. TDW gives you many ways to control your program's execution. You can

- execute single machine instructions or single source lines
- skip over calls to functions or procedures
- "animate" the debugger (perform continuous tracing)
- run until the current function or procedure returns to its caller
- run to a specified location
- continue until a breakpoint is reached
- reverse program execution

A debugging session consists of alternating periods when either your program or the debugger is running. When the debugger is running, you can cause your program to run by choosing one of the Run menu's command options or pressing its hot key equivalent. When your program is running, the debugger starts up again when either the specified section of your program has been executed, or you interrupt execution with a special key sequence, or TDW encounters a breakpoint.

This chapter shows you how to examine the state of your program whenever TDW is in control. You'll see various ways to execute portions of your program, and also how to interrupt your program while it's running. Finally, you'll learn the ways you can restart a debugging session, either with the same program or with a different program.

Examining the current program state

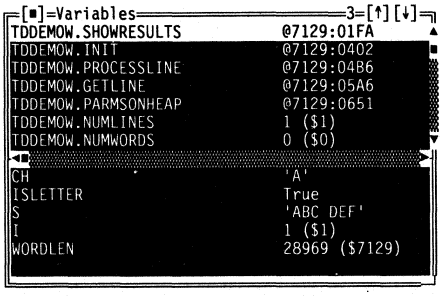
The "state" of your program consists of the following elements:

- its command-line arguments
- the stack of active functions or procedures
- the current location in the source code or machine code
- register values
- the contents of memory
- the reason the debugger stopped your program
- the value of your program data variables

The following sections explain how to use the Variables window, the Stack window, the local menus of the Global and Static panes, and the Origin and Get Info commands. See Chapter 6 for more information on how to examine and change the values of your program data variables.

The Variables window

You open the Variables window by choosing View | Variables. This window shows you all the variables (names and values) that are accessible from the current location in your program. Use it to find variables whose names you can't remember. You can then use the local menu commands to further examine or change their values. You can also use this window to examine the variables local to any function that has been called.



```
[■]=Variables                              3=[↑][↓]
TDDEMOW.SHOWRESULTS            @7129:01FA
TDDEMOW.INIT                   @7129:0402
TDDEMOW.PROCESSLINE            @7129:04B6
TDDEMOW.GETLINE                @7129:05A6
TDDEMOW.PARMSONHEAP            @7129:0651
TDDEMOW.NUMLINES               1 ($1)
TDDEMOW.NUMWORDS               0 ($0)
CH                             'A'
ISLETTER                       True
S                              'ABC DEF'
I                              1 ($1)
WORDLEN                        28969 ($7129)
```

Figure 5.1
The Variables window

The Variables window has two panes:

- The Global pane (top) shows all the global symbols in your program.
- The Static pane (bottom) shows all the static symbols in the current module (the module containing the current program location, CS:IP) and all the symbols local to the current function.

Both panes show the name of the variable at the left margin and its value at the right margin. If TDW can't find any data type information for the symbol, it displays four question marks (????).

Press *Alt-F10* (as with all local menus) to pop up the Global pane's local menu. If control-key shortcuts are enabled, you can also press *Ctrl* with the first letter of the desired command to access it.

If your program contains routines that perform recursive calls, or if you want to view the variables local to a function that has been called, you can examine the value of a specific instance of a function's local data. First create a Stack window with View | Stack, then move the highlight to the desired instance of the function call. Next, press *Alt-F10* and choose Locals. The Static pane of the Variables window then shows the values for that specific instance of the function.

The Global pane local menu

This local menu consists of three commands: Inspect, Change, and Watch.

Inspect

```
Inspect
Change...
Watch
```

Opens an Inspector window that shows you the contents of the currently highlighted global symbol.

If the variable you want to inspect is the name of a function, you are shown the source code for that function, or if there is no source file, a CPU window shows you the disassembled code.

See Chapter 6 for more information on how Inspector windows behave.

If the variable you inspect has a name that is superseded by a local variable with the same name, you'll see the actual value of the global variable, not the local one. This characteristic is slightly different than the usual behavior of Inspector windows, which normally show you the value of a variable from the point of view of your current program location (CS:IP). This difference gives you a convenient way of looking at the value of global variables whose names are also used as local variables.

Change

See Chapter 9 for more information on assignment and data type conversion.

Changes the value of the currently selected (highlighted) global symbol to the value you enter in the Change dialog box. TDW performs any necessary data type conversion exactly as if the assignment operator for your current language had been used to change the variable.

You can also change the value of the currently highlighted symbol by opening the Inspector window and typing a new value. When you do this, the same dialog box appears as if you had first specified the Change command.

Watch

See Chapter 6 for more information on the Watches window.

Opens a Watches window and puts the currently selected (highlighted) global symbol in the window. This command simply puts a character string in the Watches window.

The Watches window doesn't keep track of whether the variable is local or global. If you insert a global variable using the Watch command and later encounter a local variable by the same name, the local variable takes precedence as long as you are in the local variable's block. In other words, the Watches window always shows you the value of a variable from the point of view of your current program location (CS:IP).

The Static pane local menu

Press the *Alt-F10* key combination to pop up the Static pane's local menu; if control-key shortcuts are enabled, use the *Ctrl* key with the first letter of the desired command to access it.

```
Inspect
Change...
Watch
Show...
```

The Static pane has four local menu commands: Inspect, Change, Watch, and Show.

Inspect

See Chapter 6 for more information on how Inspector windows behave.

Opens an Inspector window that displays the contents of the currently highlighted module's local symbol.

Change

See Chapter 9 for more information on assignment and data type conversion.

Changes the value of the currently selected (highlighted) local symbol to the value you enter in the Change dialog box. TDW performs any data type conversion necessary, exactly as if the assignment operator had been used to change the variable.

You can also change the value of the currently highlighted symbol by opening the Inspector window (see previous command) and starting to type a new value. When you do this, the same dialog box appears as if you had first specified the Change command.

Watch

See Chapter 6 for more information on how Watches windows behave.

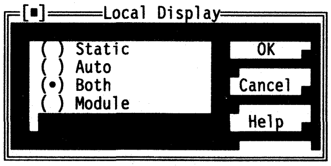
The Watch command opens a Watches window and puts the currently selected (highlighted) static or local symbol in the window.

Show

Choosing Show brings up the Local Display dialog box, which enables you to change both the scope of the variables being shown (static, auto, or both) and the module from which these variables are selected.

The following radio buttons appear in this dialog box:

Static	Show only static variables.
Auto	Show only variables local to the current block.
Both	Show both types of variables (the default).
Module	Change the current module. Brings up a dialog box showing the list of modules for the program, from which you can select a new module.



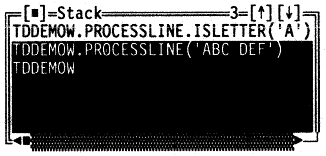
```
( ) Static
( ) Auto
(•) Both
( ) Module
```

Figure 5.2
The Local Display dialog box

The Stack window

You create a Stack window by choosing View | Stack. The Stack window lists all active functions or procedures. The most recently called routine is displayed first, followed by its caller and the previous caller, all the way back to the main program. For each procedure or function, you see the value of each parameter it was called with.

Figure 5.3
The Stack window



```
[■]=Stack=====================3=[↑][↓]
TDDEMOW.PROCESSLINE.ISLETTER('A')
TDDEMOW.PROCESSLINE('ABC DEF')
TDDEMOW
```

OOP

The Stack window likewise displays the names of object methods, each prefixed with the name of the object type that defines the method:

```
SHAPES.ACIRCLE(174, 360, 75.0) {Turbo Pascal}
```

Press *Alt-F10* to pop up the Stack window local menu, or press *Ctrl* with the first letter of the desired command to access it.

The Stack window local menu

```
Inspect
Locals
```

The Stack window local menu has two commands: Inspect and Locals.

Inspect

Opens a Module window positioned at the active line in the currently highlighted procedure. If the highlighted procedure is the top (most recently called) procedure, the Module window shows the current program location (CS:IP). If the highlighted procedure is one of the procedures that called the most recent procedure, the cursor is positioned on the line in the procedure that will be executed after the called procedure returns.

You can also invoke this command by positioning the highlight bar over a procedure, then pressing *Enter.*

Locals

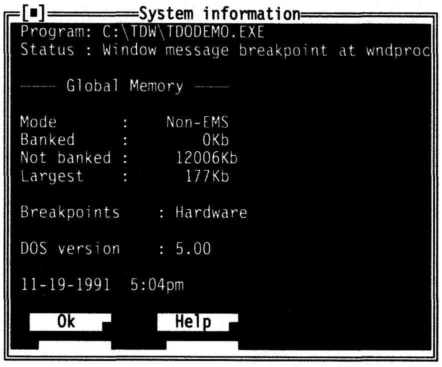
Opens a Variables window that shows the symbols local to the current module and the symbols local to the currently highlighted procedure. If a procedure calls itself recursively, there are multiple instances of the procedure in the Stack window. By positioning the highlight bar on one instance of the procedure, you can use this command to look at the local variables in that instance.

The Origin local menu command

Both the Module window and the Code pane of a CPU window have an Origin command on their local menus. Origin positions the cursor at the current code segment (CS:IP). This is very useful when you have been looking at your code and want to get back to where your program stopped.

The Get Info command

You can choose File | Get Info to look at memory use and to determine why the debugger gained control. This command produces a text box that disappears when you press *Enter*, *Spacebar*, or *Esc*.



```
Program: C:\TDW\TDODEMO.EXE
Status : Window message breakpoint at wndproc

---- Global Memory ----

Mode        :    Non-EMS
Banked      :        0Kb
Not banked  :    12006Kb
Largest     :      177Kb

Breakpoints     : Hardware

DOS version     : 5.00

11-19-1991  5:04pm
```

Figure 5.4
The Get Info text box

The following information appears in the System Information box:

- The name of the program you're debugging
- A description of why your program stopped
- Information about the global memory on your system
- The DOS version you're running
- The current date and time

Global memory information

TDW provides you with the following information about global memory:

Mode Memory modes can be large-frame EMS, small-frame EMS, and non-EMS (extended memory).

Banked	The amount in kilobytes of memory above the EMS bank line (eligible to be swapped to expanded memory if the system is using it).
Not banked	The amount in kilobytes of memory below the EMS bank line (not eligible to be swapped to expanded memory).
Largest	The largest contiguous free block of memory, in kilobytes.

Status line messages

Here are the messages you'll see on the second (status) line, describing why your program stopped:

Stopped at __

Your program stopped as the result of a completed Run | Execute To, Run | Go to Cursor, or Run | Until Return command. This status line message also appears when your program is first loaded, and the compiler startup code in your program has been executed to put you at the start of your source code.

No program loaded

You started TDW without loading a program. You cannot execute any code until you either load a program or assemble some instructions using the Assemble local menu command in the Code pane of a CPU window.

Trace

You executed a single source line or machine instruction with *F7* (Run | Trace).

Step

You executed a single source line or machine instruction, skipping procedure and function calls, with *F8* (Run | Step Over).

Breakpoint at __

Your program encountered a breakpoint that was set to stop your program. The text after "at" is the address in your program where the breakpoint occurs.

Window message breakpoint at __

Your program encountered a Windows message breakpoint that was set to stop your program. The text after "at" is the window procedure the message was destined for.

Terminated, exit code __

Your program has finished executing. The text after "code" is the numeric exit code returned to Windows by your program. If your program does not explicitly return a value, a garbage value might be displayed. You cannot run your program until you reload it with Run | Program Reset.

Loaded

You either reset your program or loaded TDW and specified both a program and the option that prevents the compiler startup code from executing. Because no instructions have been executed at this point, including those that set up your stack and segment registers, if you try to examine certain data in your program, you might see incorrect values.

Interrupt

You pressed the interrupt key (*Ctrl-Alt-SysRq*) to regain control. Your program was interrupted and control passed back to the debugger.

Exception __

A processor exception has occurred, which usually happens when your program attempts to execute an illegal instruction opcode. The Intel processor documentation describes each exception code in complete detail.

The most common exception to occur with a Windows program is Exception 13. This exception indicates that your program has attempted to perform an invalid memory access. (Either the selector value in a segment register is invalid or the offset portion of an address points beyond the end of the segment.) You must correct the invalid pointer causing the problem.

Divide by zero

Your program has executed a divide instruction where the divisor is zero.

Global breakpoint __ at __

A global breakpoint has been triggered. You are told the breakpoint number and the location in your program where the breakpoint occurred.

The Run menu

The Run menu has a number of options for executing different parts of your program. Since you use these options frequently, most are available on function keys.

```
Run                     F9
Go to cursor            F4
Trace into              F7
Step over               F8
Execute to...       Alt-F9
Until return        Alt-F8
Animate...
Back trace          Alt-F4
Instruction trace   Alt-F7
--------------------------
Arguments...
Program reset      Ctrl-F2
```

Run

F9

Runs your program at full speed. Control returns to TDW when one of the following events occurs:

- Your program terminates.
- A breakpoint with a break action is encountered.
- You interrupt execution with *Ctrl-Alt-SysRq*.

Go to Cursor

F4

Executes your program up to the line that the cursor is on in the current Module window or CPU Code pane. If the current window is a Module window, the cursor must be on a line of source code.

Trace Into

F7

Executes a single source line or assembly level instruction. If the current window is a Module window, a single line of source code is executed; if it's a CPU window, a single machine instruction. If the current line contains any procedure or function calls, TDW traces into the routine. If the current window is a CPU window, pressing *F7* on a CALL instruction steps to the routine being called.

OOP

TDW treats object methods just like any other procedure or function. *F7* traces into the source code if it's available.

Step Over

F8

Executes a single source line or machine instruction, skipping over any procedure or function calls. If the current window is a Module window, this command usually executes a single source line. If the current window is a CPU window, pressing *F8* on a CALL instruction steps over the routine being called.

If you step over a single source line, TDW treats any function or procedure calls in that line as part of the line. You don't end up at the start of one of the functions or procedures. Instead, you end up at the next line in the current routine or at the previous routine that called the current one.

If you are in a CPU window, TDW treats çertain instructions as a single instruction, even when they cause multiple assembly instructions to be executed. Here is a complete list of the instructions TDW treats as single instructions:

CALL	Subroutine call, near, and far
INT	Interrupt call
LOOP	Loop control with CX counter
LOOPZ	Loop control with CX counter
LOOPNZ	Loop control with CX counter

Also stepped over are **REP**, **REPNZ**, or **REPZ** followed by **CMPS**, **CMPS**, **CMPSW**, **LODSB**, **LODSW**, **MOVS**, **MOVSB**, **MOVSW**, **SCAS**, **SCASB**, **SCASW**, **STOS**, **STOSB**, or **STOSW**.

OOP

The Run | Step Over command treats a call to an object method like a single statement and steps over it like any other procedure or function call.

Execute To

Alt F9

Executes your program until the address you specify in the dialog box is reached. The address you specify might never be reached if a breakpoint action is encountered first or you interrupt execution.

Until Return

Alt F8

Executes until the current procedure returns to its caller. This is useful in two circumstances: When you have accidentally executed into a function or procedure that you are not interested in with Run | Trace instead of Run | Step, or when you have determined that the current procedure works to your satisfaction, and you don't want to slowly step through the rest of it.

Animate

Performs a continuous series of Trace Into commands, updating the screen after each one. (The effect is to run your program in slow motion.) You can watch the current location in your source code and see the values of variables changing. Press any key to interrupt this command.

After you choose Run | Animate, TDW prompts you for a time delay between successive traces. The time delay is measured in tenths of a second; the default is 3.

Back Trace

Alt F4

Some restrictions apply to using the Execution History window. See page 86 for more information.

If you are tracing (*F7* or *Alt-F7*) through your program, Back Trace reverses the order of execution. Reverse execution is handy if you trace beyond the point where you think there may be a bug, and want to reverse program execution back to that point. This feature lets you "undo" the execution of your program by stepping backward through the code, either a single step at a time or to a specified point highlighted in the Execution History window.Reverse execution is always available in the CPU window. However, you can only execute source code in reverse if full history is *On*. (Use the View | Execution History command to bring up the Execution History window, then in the local menu set Full History *On*.)

TDW will not execute in reverse any Windows code called by your program unless you are in the CPU window and the code is in a DLL you have selected for debugging.

Instruction Trace

Alt F7

Executes a single machine instruction. Use this command when you want to trace into an interrupt, or when you're in a Module window and you want to trace into a procedure or function that's in a module with no debug information (for example, a library routine).

Since you will no longer be at the start of a source line, this command usually places you in a CPU window.

Arguments

This command lets you set new command-line arguments for your program. For a discussion of this command, see "Changing the program arguments" on page 90.

Program Reset

Reloads from disk the program you're debugging. You might use this command

- When you've executed past the place where you think there is a bug.
- When your program has terminated and you want to run it again.
- If you're in a Module or CPU window, you've suspended your Windows application program with *Ctrl-Alt-SysRq,* and you want to terminate it and start over.
- If you've already loaded your application, you've just set startup debugging for one or more dynamic link libraries (DLLs), and you now want to debug those DLLs.

Ctrl F2

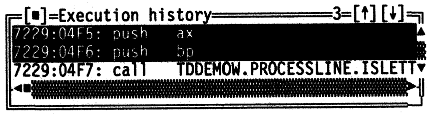
If you're in a Module or CPU window, the debugger sets the current-line marker at the start of the program, but the display stays exactly where you were when you chose Program Reset. This behavior makes it easier for you to set the cursor near where you were and run the program to that line.

If you chose Program Reset because you just executed one source statement more than you intended, you can position the cursor up a few lines in your source file and press *F4* to run to that location. Alternatively, if Full History had been on (see the local menu of the View | Execution History window), you could have chosen Run | Back Trace to step back through previously executed code instead of choosing Program Reset.

The Execution History window

TDW has a special feature called *execution history* that keeps track of each instruction as it's executed (provided that you're tracing into the code). You can examine these instructions and, if you like, undo them to return to a point in the program where you think there might be a bug. TDW can record about 400 instructions.

Figure 5.5
The Execution History window



```
[■]=Execution history═══════════════3=[↑][↓]
7229:04F5: push   ax
7229:04F6: push   bp
7229:04F7: call   TDDEMOW.PROCESSLINE.ISLETT
```

You can examine the execution history in the Execution History window, which you open by choosing View | Execution History.

The Execution History window shows instructions already executed that you can examine or undo. Use the highlight bar to make your selection.

The execution history only keeps track of instructions that have been executed with the Trace Into command (*F7*) or the Instruction Trace command (*Alt-F7*). It also tracks for Step Over, as long as you don't encounter one of the commands listed on page 83 or 87. As soon as you use the Run command or execute an interrupt, the execution history is deleted. (It starts being recorded again when you go back to tracing.)

You cannot backtrace into an interrupt call.

If you step over a procedure or function call, you will not be able to trace back beyond the instruction following the return.

Backtracing through a port-related instruction has no effect, since you can't undo reads and writes.

The local menu

```
Inspect
Reverse execute
──────────────────────
Full history        No
```

The local menu for the Instructions pane contains three options, Inspect, Reverse Execute, and Full History.

Inspect

This command takes you to the command highlighted in the Instructions pane. If it is a line of source code, you are shown that line in the Module window; if there is no source code, the CPU window opens, with the instruction highlighted in the Code pane.

Reverse Execute

Alt F4

This command reverses program execution to the location highlighted in the Instructions pane. If you selected a line of source code, you are returned to the Module window; otherwise, the CPU window appears with the highlight bar of the Code pane on the instruction.

Warning! You can never reverse back over a section of your program that you didn't trace through. For example, if you set a breakpoint and then pressed *F9* to run until the breakpoint was reached, all your reverse execution history will be thrown away.

Warning! The **INT** instruction causes any previous execution history to be thrown out. You can't reverse back over this instruction, unless you press *Alt-F7* to trace into the interrupt.

The following instructions do not cause the history to be thrown out, but they cannot have their effects undone. You should look out for unexpected side effects if you back up over these instructions:

IN	**INSW**
OUT	**OUTSB**
INSB	**OUTSW**

Full History

This command is a toggle. If it is set to *On*, backtracing is enabled. If it is *Off*, backtracing is disabled.

Interrupting program execution

Because Windows applications are interactive programs, the best way to debug one is to run the application and then interrupt it or cause it to encounter a breakpoint.

As a primary debugging technique, stepping or tracing through a Windows application can be of marginal utility because eventually you reach code that sits in a loop, waiting for a message for a window. Instead, you should set code and message breakpoints if possible, run your program until it encounters one of these breakpoints, and then step or trace if necessary.

If you do step into the message loop, you can press the *Alt-F5* key combination to see the application screen, but you won't be able to interact with the program. Instead, you can press *F9* to run the program so you can use the application's windows. But what happens if you need to get back to TDW to track down a bug that shows up while you're using one of your application's windows?

What you can do is interrupt your program by pressing the *Ctrl-Alt-SysRq* key combination. Once you're back in TDW, you can set code or message breakpoints, set up views, look at any messages you might have been logging, or whatever else you need to do to track the bug. When you're ready to return to the application again, press *F9* to run it.

When you return to TDW, if you see a CPU window without any lines corresponding to lines in your code, you're probably in Windows code. You can display the Module window and set breakpoints or whatever else you need to do, but there are some things you *should not* do:

- Single-step through your program. Attempting to single-step after interrupting your application can have unpredictable effects if your application was executing Windows code. A typical result is that Windows terminates both your application and TDW, generating the message, "Unrecoverable application error."
- Terminate or reload either your application or TDW. If you do, Windows gets confused and hangs, forcing you to reboot. If you do try to exit or reload in this situation, TDW displays the following prompt in a dialog box:

  ```
  Ctrl-Alt-SysRq interrupt, system crash possible, Continue?
  ```

  At this point, the best course of action is to select *No*, return to TDW and set a breakpoint you know your code will hit, then run your application again and cause it to hit the breakpoint and exit to TDW.

Program termination

When your program terminates and exits back to Windows, TDW regains control. It displays a message showing the exit code that your program returned to Windows. Once your program terminates, using any of the Run menu options causes TDW to reload your program.

The segment registers and stack are usually not correct when your program has terminated, so do not examine or modify any program variables after termination.

Restarting a debugging session

TDW has a feature that makes restarting a debugging session as painless as possible. When you're debugging a program, it's easy to go just a little too far and overshoot the real cause of the problem. In that case, TDW lets you restart debugging but suspends execution before the last few commands that caused you to miss the problem that you wanted to observe. How? It lets you reload your last program from disk, and preserves any previous command-line arguments.

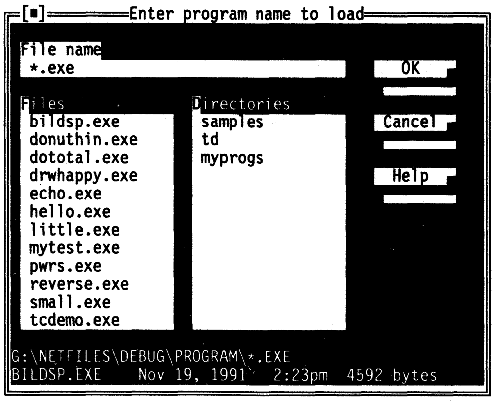
To reload the program you were debugging, choose Run | Program Reset (*Ctrl-F2*). TDW reloads the program from disk, with any data you have added since you last saved to disk. Reloading is the safest way to restart a program. Restarting by executing at the start of the program can be risky, since many programs expect certain data to be initialized from the disk image of the program.

Program Reset leaves breakpoints and watchpoints intact.

Opening a new program to debug

You load a new program to debug by choosing File | Open to open the Enter Program Name to Load dialog box.

Figure 5.6
The Enter Program Name to Load dialog box



```
*.exe

bildsp.exe        samples
donuthin.exe      td
dototal.exe       myprogs
drwhappy.exe
echo.exe
hello.exe
little.exe
mytest.exe
pwrs.exe
reverse.exe
small.exe
tcdemo.exe

G:\NETFILES\DEBUG\PROGRAM\*.EXE
BILDSP.EXE    Nov 19, 1991   2:23pm   4592 bytes
```

You can enter a file name (extension .EXE) in the File Name input box, or press *Enter* to activate a list box of all the .EXE files in the current directory. Move the highlight bar to the file you want to load and press *Enter*.

Another way of specifying a file in the list box is to type in the name of the file you want to load. The highlight bar in the Files list box moves to the file that begins with the first letter(s) you typed. When the bar is positioned on the file you want, press *Enter*.

You can supply arguments to the program to debug by placing them after the program name, as follows:

```
myprog a b c
```

This command loads program *MyProg* with three command-line arguments, *a*, *b*, and *c*.

Changing the program arguments

If you forgot to supply some necessary arguments to your program when you loaded it, you can use the Run | Arguments command to set or change the arguments. Enter new arguments exactly as you would following the name of your program on the command line.

Once you have entered new arguments, TDW asks you if you want to reload your program from disk. You should answer Yes, because for most programs, the new arguments will only take effect if you reload the program first.

C H A P T E R

6

Examining and modifying data

TDW provides a unique and intuitive way to examine and even change your program's data.

- Inspector windows let you look at your data as it appears in your source file. You can "follow" pointers, scroll through arrays, and see structures, records, and unions exactly as you wrote them.
- You can also put variables and expressions into the Watches window, where you can watch their values as your program executes.
- The Evaluate/Modify dialog box shows you the contents of any variable and lets you assign a new value to it.

This chapter assumes that you understand the various data types that can be used in Turbo Pascal for Windows. If you are fairly new to the language and have not yet explored its simple data types (Char, Integer, Boolean, Word, Real, String, Longint, Shortint, and so on), it would be helpful to first learn about them before reading the chapter.

If you're familiar with the simple data types, parts of this chapter are useful. When you've delved into the more complex data types (arrays, pointers, PChars, records, files, sets, objects, and so on), you can return to this chapter to learn more about looking at them with TDW.

For how to examine or modify arbitrary blocks of memory as hex data bytes, see Chapter 12.

This chapter shows you how to examine and modify variables in your program. First, we explain the Data menu and its options. We then show you how to point directly at data items in your source modules. Finally, we introduce the Watches window and describe the way that the data types appear in Inspector windows.

The Data menu

```
Inspect...
Evaluate/modify... Ctrl-F4
Add watch...       Ctrl-F7
Function return
```

The Data menu lets you choose how to examine and change program data. You can evaluate an expression, change the value of a variable, and open Inspector windows to display the contents of your variables.

Inspect

Prompts you for the variable that references the data you want to inspect, then opens an Inspector window that shows the contents of the program variable or expression. You can enter a simple variable name or a complex expression.

If the cursor is on a variable in a text pane when you issue this command, the dialog box automatically contains the variable, if any, at the cursor. If you select an expression in a text pane (using *Ins*), the dialog box contains the selected expression.

Inspector windows really come into their own when you want to examine a complicated data structure, such as an array of records or a linked list of items. Since you can inspect items within an Inspector window, you can "walk" through your program's data objects as easily as you scroll through your source code in the Module window.

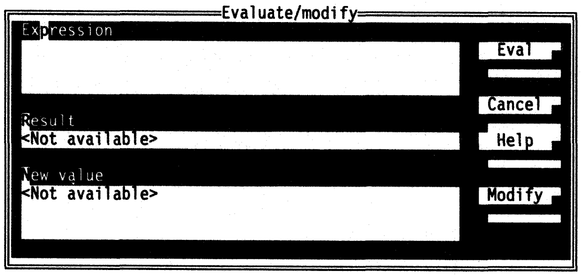
See the "Inspector windows" section later in this chapter for a complete description of how Inspector windows behave.

Evaluate/Modify

See Chapter 9 for a complete discussion of expressions.

Opens the Evaluate/Modify dialog box (Figure 6.1), which prompts you for an expression to evaluate, then evaluates it exactly as the compiler would during compilation when you choose the Eval button.

If the cursor is in a text pane when you issue this command, the dialog box automatically contains the variable, if any, at the cursor. If you select an expression (using *Ins*), the dialog box contains the marked expression.



```
<Not available>
<Not available>
```

Figure 6.1
The Evaluate/Modify dialog box

See Chapter 9 for a discussion of format control.

Remember that you can add a format control string after the expression you want to watch. TDW displays the result in a format suitable for the data type of the result. To display the result in a different format, put a comma (,) separator, then a format control string after the expression. Displaying in a different format is useful when you want to watch something, but your program displays it in a format other than TDW's default display format for the data type.

The dialog box has three fields.

- In the top field, you type the expression you want to evaluate. This field is the Evaluate input box, and it has a history list just like any other input box.
- The middle field displays the result of evaluating your expression.
- The bottom field is an input box where you can enter a new value for the expression. If the expression can't be modified, this box reads `<Not available>`, and you can't move your cursor into it.

Your entry in the New Value input box takes effect when you choose the Modify button. Use *Tab* and *Shift-Tab* to move from one box to another, just as you do in other dialog boxes. Press *Esc* from inside any input box to remove the dialog box, or click the Cancel button with your mouse.

Data strings too long to display in the Result input box are terminated by an arrow (▸). You can see more of the string by scrolling to the right.

OOP

If you're debugging an object-oriented Pascal program, the Evaluate/Modify dialog box also lets you display the fields of an object instance. You can use any format specifier with an instance that can be used in evaluating a record.

When you're tracing inside a method, TDW knows about the scope and presence of the *Self* parameter. You can evaluate *Self* and follow it with format specifiers and qualifiers.

You cannot execute constructor or destructor methods in the Evaluate window.

TDW also lets you call a method from inside the Evaluate/Modify dialog box. Just type the instance name followed by a dot, followed by the method name, followed by the actual parameters (or empty parentheses if there are no parameters). With these declarations,

```
type
  Point = object
    X, Y    : Integer;
    Visible : Boolean;
    constructor Init(InitX, InitY : Integer);
    destructor  Done; virtual;
    procedure   Show; virtual;
    procedure   Hide; virtual;
    procedure   MoveTo(NewX, NewY : Integer);
end;

var
  APoint : Point;
```

you could enter any of these expressions in TDW's Evaluate window:

Expression	Result
APoint.X	5 ($5) : Integer
APoint	(5,23,FALSE) : Point
APoint.MoveTo	@6F4F : 00BE
APoint.MoveTo(10, 10)	Calls method *MoveTo*
APoint.Show()	Calls method *Show*

You can also use the Evaluate/Modify dialog box as a simple calculator by typing in numbers as operands instead of program variables.

Add Watch

Prompts you for an expression to watch, then places the expression or program variable on the list of variables displayed in the Watches window when you press *Enter* or choose the OK button.

If the cursor is in a text pane when you issue this command, the dialog box automatically contains the variable at the cursor, if any. If you select an expression (using *Ins*), the dialog box contains the selected expression.

Function Return

Shows you the value the current procedure is about to return. Use this command only when the procedure is about to return to its caller.

The return value is displayed in an Inspector window, so you can easily examine return values that are pointers to compound data objects.

Function Return saves you from having to switch to a CPU window to examine the return value placed in the CPU registers. And since TDW also knows the data type being returned and formats it appropriately, this command is much easier to use than a hex dump.

Pointing at data objects in source files

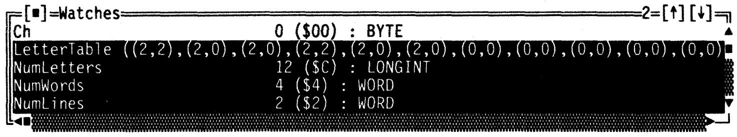
See Chapter 8 for a full discussion of using Module windows.

TDW has a powerful mechanism to relieve you from always typing in the names of program variables that you want to inspect. From within any Module window, you can place the cursor anywhere within a variable name and use the local menu Inspect command to create an Inspector window showing the contents of that variable. You can also select an expression or variable to inspect by pressing *Ins* and using the cursor keys to highlight it before choosing Inspect.

The Watches window

The Watches window lets you list variables and expressions in your program whose values you want to track. You can watch the value of both simple variables (such as integers) and complex data objects (such as arrays). In addition, you can watch the value of a calculated expression that does not refer directly to a memory location. For example, $x * y + 4$.

Figure 6.2
The Watches window



```
[■]=Watches                                                      2=[↑][↓]
Ch                          0 ($00) : BYTE
LetterTable ((2,2),(2,0),(2,0),(2,2),(2,0),(2,0),(0,0),(0,0),(0,0),(0,0),(0,0)
NumLetters                  12 ($C) : LONGINT
NumWords                    4 ($4) : WORD
NumLines                    2 ($2) : WORD
```

Choose View I Watches to access the Watches window. It holds a list of variables or expressions whose values you want to watch. For each item, the variable name or expression appears on the left and its data type and value on the right. Compound values like arrays and records appear with their values between parentheses. If there isn't room to display the entire name or expression, it is truncated.

See Chapter 9 for a complete discussion of scopes and when a variable or parameter is valid.

When you enter an expression to watch, you can use variable names that are not valid yet because they are in a procedure or function that hasn't been called. TDW lets you set up a watch expression before its scope becomes active. This situation is the only time you can enter an expression that can't be immediately evaluated.

Warning!

If you mistype the name of a variable, the mistake won't be detected because TDW assumes it is the name of a variable that will become available as your program executes.

Unless you use the scope-overriding mechanism discussed in Chapter 9, TDW evaluates an expression in the Watches window in the scope of the current location where your program is stopped, as if the expression appeared in your program at that place. If a watch expression contains a variable name that is not accessible from the current scope—for example, if it's private to another module—the value of the expression is undefined and is displayed as four question marks (????).

OOP

When you're tracing inside an object method, you can add the *Self* parameter to the Watches window.

The Watches window local menu

As with all local menus, press *Alt-F10* to pop up the Watches window local menu. If you have control-key shortcuts enabled, press *Ctrl* with the first letter of the desired command to access it.

```
Watch...
Edit...
Remove
Delete all
-----------
Inspect
Change...
```

Watch

Prompts you for the variable name or expression to add to the Watches window. It is added at the current cursor location.

Edit

Opens a dialog box in which you can edit an expression in the Watches window. You can change any watch expression that's there, or enter a new one.

You can also invoke this command by pressing *Enter* once you've positioned the highlight bar over the watch expression you want to change. Press *Enter* or choose the OK button to put the edited expression into the Watches window.

Remove

Removes the currently selected item from the Watches window.

Delete All

Removes all the items from the Watches window. This command is useful if you move from one area of your program to another, and the variables you were watching are no longer relevant. (Then use the Watch command to enter more variables.)

Inspect

Opens an Inspector window to show you the contents of the currently highlighted item in the Watches window. If the item is a compound object (array or record), you can view all its elements, not just the ones that fit in the Watches window.

Change

See Chapter 9 for more information on the assignment operator and type conversion (casting).

Changes the value of the currently highlighted item in the Watches window to the value you enter in the dialog box. If the current language you're using permits it, TDW performs any necessary type conversion exactly as if the assignment operator had been used to change the variable.

Inspector windows

An Inspector window displays your program data appropriately, depending on the data type you're inspecting. Inspector windows behave differently for scalars (for example, Char or Integer), pointers (^), arrays (**array** [1..10] **of** Word), functions, records, and sets.

The Inspector window lists the items that make up the data object being inspected. The title of the window shows the expression or the name of the variable being inspected.

The first item in an Inspector window is always the memory address of the data item being inspected, expressed as a segment: offset pair, unless it has been optimized to a register or is a constant (for example, 3).

To examine the contents of a variable in an Inspector window as raw data bytes, choose View | Dump while you're in the Inspector window. The Dump window comes up, with the cursor positioned to the data displayed in the Inspector window. You can return to the Inspector window by closing the window with the Window | Close command (*Alt-F3*), or clicking the close box with your mouse.

The following sections describe the different Inspector windows that can appear for two of the languages supported by TDW: Pascal and assembler. The programming language used dictates the format of the information displayed in Inspector windows. Data items and their values always appear in a format similar to the one they were declared with in the source file.

Remember that you don't have to do anything special to cause the different Inspector windows to appear. The right one appears automatically, depending on the data you're inspecting.

Data Inspector windows

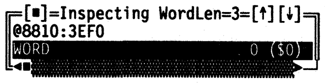
Scalars

Scalar Inspector windows show you the value of simple data items, such as

```
var
  X : Integer;
  Y : Longint;
```

These Inspector windows have only a single line of information following the top line that gives the address of the variable. To the left appears the type of the scalar variable (Byte, Word, Integer, Longint, and so forth), and to the right appears its present value. The value can be displayed as decimal, hex, or both. It's usually displayed first in decimal, with the hex values in parentheses (using the Turbo Pascal hex prefix $). You can use TDWINST to change how the value is displayed.

If the variable being displayed is of type Char, the character equivalent is also displayed. If the present value does not have a printing character equivalent, TDW uses a pound sign (#) followed by a number to display the character value. This character value appears before the decimal or hex values.



```
[■]=Inspecting WordLen=3=[↑][↓]
@8810:3EF0
WORD                     0 ($0)
```

Figure 6.3
A Pascal scalar Inspector window

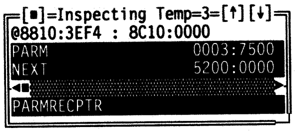
Pointers Pointer Inspector windows in a Pascal program show you the value of data items that point to other data items, such as

```
var
  IP : ^integer;
  LP : ^^pointer;
```

Pointer Inspector windows usually have only a single line of information following the top line that gives the address of the variable. To the left appears [1], indicating the first member of an array. To the right appears the value of the item being pointed to. If the value is a complex data item such as a record or an array, however, only as much of it as possible is displayed, with the values enclosed in parentheses.

If the pointer is of type **PChar** and appears to be pointing to a null-terminated character string, more information appears showing the value of each item in the character array. To the left in each line appears the array index ([1], [2], and so on), and the value appears to the right as it would in a scalar Inspector window. In this case, the entire string is also displayed on the top line, along with the address of the pointer variable and the address of the string that it points to.

You also get multiple lines if you open the Inspector window and issue the Range local menu command, specifying a count greater than 1.



```
[■]=Inspecting Temp=3=[↑][↓]
@8810:3EF4 : 8C10:0000
PARM                0003:7500
NEXT                5200:0000
PARMRECPTR
```

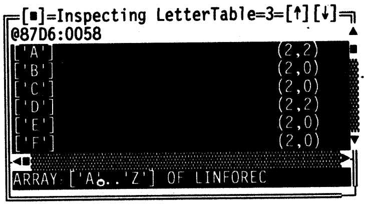
Figure 6.4
A Pascal pointer Inspector window

Arrays Array Inspector windows in Pascal programs show you the value of arrays of data items, such as

```
var
  A : array[1..10,1..20] of Integer;
  B : array[1..50] of Boolean;
```

There is a line for each member of the array. To the left on each line appears the array index of the item and to the right is its present value. If the value is a complex data item such as a record or an array, as much of it as possible is displayed, with the values enclosed in parentheses.

You can use the Range command to examine any portion of an array. This is useful if the array has a lot of elements, and you want to look at something in the middle of the array.



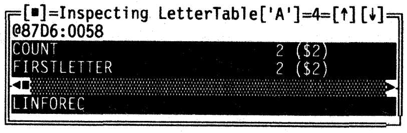
```
[■]=Inspecting LetterTable=3=[↑][↓]
@87D6:0058
['A']                        (2,2)
['B']                        (2,0)
['C']                        (2,0)
['D']                        (2,2)
['E']                        (2,0)
['F']                        (2,0)
ARRAY ['A'..'Z'] OF LINFOREC
```

Figure 6.5
A Pascal array Inspector window

Records Record Inspector windows in Pascal programs show you the value of the fields in your records. For example,

```
record
  year  : Integer;
  month : 1..12;
  day   : 1..31;
end
```

These Inspector windows have another pane below the one that shows the values of the fields. This additional pane shows the data type of the field highlighted in the top pane.

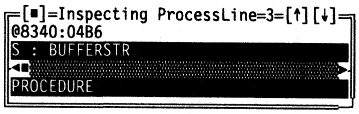


```
[■]=Inspecting LetterTable['A']=4=[↑][↓]
@87D6:0058
COUNT                       2 ($2)
FIRSTLETTER                 2 ($2)

LINFOREC
```

Figure 6.6
A Pascal record Inspector window

Procedures and functions

In the upper pane, procedure and function Inspector windows in Pascal programs give you information about calling parameters. These windows have a second pane in which the routine is identified as a procedure or function. If the routine is a function, you also see the data type it returns.



```
[■]=Inspecting ProcessLine=3=[↑][↓]
@8340:04B6
S : BUFFERSTR

PROCEDURE
```

Figure 6.7
A Pascal procedure Inspector window

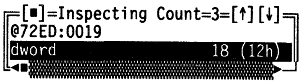
Assembler data Inspector windows

Scalars

Scalar Inspector windows in assembly language programs show you the value of simple data items, such as

```
VAR1    DW  99
MAGIC   DT  4.608
BIGNUM  DD  123456
```

These Inspector windows have only a single line of information following the top line that gives the address of the variable. To the left appears the type of the scalar variable (**BYTE**, **WORD**, **DWORD**, **QWORD**, and so forth), and to the right appears its present value. The value can be displayed as decimal, hex, or both. It's usually displayed first in decimal, with the hex values in parentheses (using the standard assembler hex postfix H). You can use TDWINST to change how the value is displayed.



```
[■]=Inspecting Count=3=[↑][↓]
@72ED:0019
dword                  18 (12h)
```

Figure 6.8
An assembler scalar Inspector window

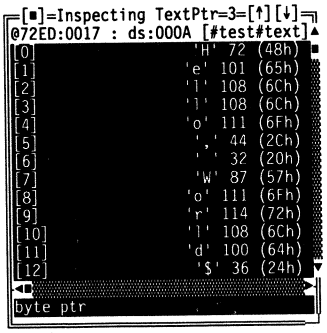
Pointers Pointer Inspector windows in assembler programs show you the value of data items that point to other data items, such as

```
X        DW  0
XPTR     DW  X
FARPTR   DD  X
```

Pointer Inspector windows usually have only a single line of information following the top line that gives the address of the variable. To the left appears [0], indicating the first member of an array. To the right appears the value of the item being pointed to. If the value is a complex data item such as an array, however, only as much of it as possible is displayed, with the values enclosed in braces ({}).

If the pointer is of type **BYTE** and appears to be pointing to a null-terminated character string, more information appears, showing the value of each item in the character array. To the left in each line appears the array index ([1], [2], and so on), and the value appears to the right as it would in a scalar Inspector window. In this case, the entire string is also displayed on the top line, along with the address of the variable and the address of the string that it points to.

You also get multiple lines if you open the Inspector window with a Range local menu command and specify a count greater than 1.



```
=[■]=Inspecting TextPtr=3=[↑][↓]=
@72ED:0017 : ds:000A [#test#text]
[0]                'H'  72 (48h)
[1]                'e' 101 (65h)
[2]                'l' 108 (6Ch)
[3]                'l' 108 (6Ch)
[4]                'o' 111 (6Fh)
[5]                ','  44 (2Ch)
[6]                ' '  32 (20h)
[7]                'W'  87 (57h)
[8]                'o' 111 (6Fh)
[9]                'r' 114 (72h)
[10]               'l' 108 (6Ch)
[11]               'd' 100 (64h)
[12]               '$'  36 (24h)
byte ptr
```

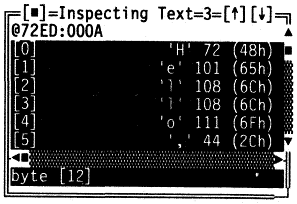
Figure 6.9
An assembler pointer Inspector window

Arrays Array Inspector windows in assembler programs show you the value of arrays of data items, such as

```
WARRAY DW 10 DUP (0)
MSG    DB "Greetings",0
```

There is a line for each member of the array. To the left on each line appears the array index of the item and to the right is its present value. If the value is a complex data item such as a **STRUC**, however, only as much of it as possible is displayed.

You can use the Range local command to examine a portion of an array. This is useful if the array has a lot of elements, and you want to look at something in the middle of the array. When you choose Range, you are prompted to enter a starting index followed by a comma and the number of members to inspect.



```
[■]=Inspecting Text=3=[↑][↓]
@72ED:000A
[0]             'H'  72 (48h)
[1]             'e' 101 (65h)
[2]             'l' 108 (6Ch)
[3]             'l' 108 (6Ch)
[4]             'o' 111 (6Fh)
[5]             ','  44 (2Ch)
byte [12]
```

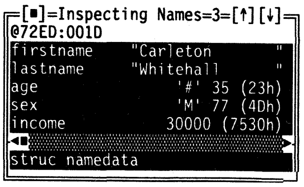
Figure 6.10
An assembler array Inspector window

Structures and unions

Structure Inspector windows in assembler programs show you the value of the fields in your **STRUC** and **UNION** data objects. For example,

```
X       STRUC
MEM1          DB   ?
MEM2          DD   ?
X       ENDS
ANX     X          <1,ANX>

Y       UNION
ASBYTES       DB   10 DUP (?)
ASFLT         DT   ?
Y       ENDS
AY      Y          <?,1.0>
```

These Inspector windows have another pane below the one that shows the values of the fields. This additional pane shows the data type of the field highlighted in the top pane.



```
[■]=Inspecting Names=3=[↑][↓]
@72ED:001D
firstname    "Carleton       "
lastname     "Whitehall      "
age                '#' 35 (23h)
sex                'M' 77 (4Dh)
income          30000 (7530h)
struc namedata
```

Figure 6.11
An assembler structure Inspector window

The Inspector window local menu

```
Range...
Change...
─────────────────
Inspect
Descend
New expression...
Type cast...
```

The commands in this menu give the Inspector window its real power. By choosing the Inspect local menu command, for example, you create another Inspector window that lets you go into your data objects. Other commands in the menu let you inspect a range of values or a new variable.

Press *Alt-F10* to pop up the Inspector window local menu. If you have control-key shortcuts enabled, press *Ctrl* with the first letter of the desired command to access it.

Range

Sets the starting element and number of elements that you want to display. Use this command when you are inspecting an array, and you only want to look at a certain subrange of all the members of the array.

If you have a long array and want to look at a few members near the middle, use this command to open the Inspector window at the array index that you want to examine.

Change

Changes the value of the currently highlighted item to the value you enter in the dialog box. If the current language permits it, TDW performs any necessary casting exactly as if the appropriate assignment operator had been used to change the variable. See Chapter 9 for more information on the assignment operator and casting.

Inspect

Opens a new Inspector window that shows you the contents of the currently highlighted item. This is useful if an item in the Inspector window contains more items itself (like a structure or array), and you want to see each of those items. If the current Inspector window is inspecting a procedure, issuing the Inspect command shows you the source code for that procedure.

You can also invoke this command by pressing *Enter* after highlighting the item you want to inspect.

You return to the previous Inspector window by pressing *Esc* to close the new Inspector window. If you are through inspecting a data structure and want to remove all the Inspector windows, use the Window | Close command or its shortcut, *Alt-F3*.

Descend

This command works like the Inspect local menu command except that instead of opening a new Inspector window to show the contents of the highlighted item, it puts the new item in the current Inspector window. This is like a hybrid of the New Expression and Inspect commands.

☞ Once you have descended into a data structure like this, you can't go back to the previous unexpanded data structure. Use this command when you want to work your way through a complicated data structure or long linked list, but you don't care about returning to a previous level of data. This helps reduce the number of Inspector windows onscreen.

New Expression

Prompts you for a variable name or expression to inspect, without creating another Inspector window. This lets you examine other data without having to put more Inspector windows on the screen. Use this command if you are no longer interested in the data in the current Inspector window.

OOP

Inspector windows for Pascal objects are somewhat different from regular Inspector windows. See Chapter 10 for a description of object type Inspector windows.

Type Cast

*Chapter 11 explains on page 175 how to use the **gh2fp** and **lh2fp** types.*

Lets you specify a different data type (Byte, Word, Integer, Char pointer, gh2fp, lh2fp) for the item being inspected. Typecasting is useful if the Inspector window contains a symbol for which there is no type information, as well as for explicitly setting the type for untyped pointers.

C H A P T E R

7

Breakpoints

TDW uses the single term "breakpoint" to refer to the group of functions that other debuggers usually call breakpoints, watchpoints, and tracepoints.

Traditionally, breakpoints, watchpoints, and tracepoints are defined like this: A *breakpoint* is a place in your program where you want execution to stop so that you can examine program variables and data structures. A *watchpoint* causes your program to be executed one instruction or source line at a time, watching for the value of an expression to become true. A *tracepoint* causes your program to be executed one instruction or source line at a time, watching for the value of certain program variables or memory-referencing expressions to change.

TDW unifies these three concepts by defining a breakpoint in three parts:

- the location in the program where the breakpoint occurs
- the condition under which the breakpoint is triggered
- the action that takes place when the breakpoint triggers

The *location* can be either a single source line in your program or it can be global in context; a global breakpoint checks the breakpoint condition after the execution of each source line or instruction in your program.

The *condition* can be

- always
- when an expression is true

- when a data object changes value
- when a Windows message comes in

See Chapter 11, page 164, for a description of how to set a message breakpoint.

A *pass count* can also be specified, requiring that a condition be true a designated number of times before the breakpoint is triggered.

The *action* taken when a breakpoint triggers can be one of the following:

- stop program execution (a breakpoint)
- log the value of an expression
- execute an expression (code splice)
- enable a group of breakpoints
- disable a group of breakpoints

In this chapter, you'll learn about the Breakpoint and Log windows; how to set simple breakpoints, conditional breakpoints, and breakpoints that log the value of your program variables; and how to set breakpoints that watch for the exact moment when a program variable, expression, or data object changes value.

When debugging, you'll often want to set a few simple breakpoints to make your program pause execution when it reaches certain locations. You can set or clear a breakpoint at any location in your program by simply placing the cursor on the source code line and pressing *F2*. You can also set a breakpoint on any line of machine code by pressing *F2* when you are pointing at an instruction in the Code pane of a CPU window.

If you have a mouse, just click either of the leftmost two columns of the line where you want to set or remove a breakpoint. (If you're in the correct column, an asterisk (*) appears in the position indicator.)

There are two ways to access the dialog boxes for setting and customizing breakpoints. The Breakpoints menu offers a quick approach for setting breakpoints, and the Breakpoints window provides a view of the breakpoints already set, and gives access to the dialog boxes that control breakpoint settings.

The Breakpoints menu

Access the Breakpoints menu at any time by pressing the *Alt-B* hot key.

```
Toggle                F2
At...             Alt-F2
Changed memory global...
Expression true global...
Hardware breakpoint...
Delete all
```

Toggle

The Toggle command sets or clears a breakpoint at the currently highlighted address in the Module or CPU window. The hot key is *F2*.

At

See page 114 for a description of the Breakpoint Options dialog box.

At lets you set a breakpoint at a specific location in your program. When selected, At opens the Breakpoint Options dialog box, from which you can set all breakpoint options. *Alt-F2* is the hot key for At.

Changed memory global

For more information, see the "Changed memory breakpoints" section on page 121.

Changed Memory Global sets a global breakpoint that's triggered when an area of memory changes value. You are prompted for the area of memory to watch with the Enter Memory Address, Count input box. The variable expression entered is checked for change each time a line of source code is executed.

Expression true global

For more information, see "Conditional expressions" on page 122.

Expression True Global sets a global breakpoint that is triggered when the value of a supplied expression is true (nonzero). You are prompted for the expression to evaluate with the Enter Expression for Condition Breakpoint input box. The expression entered is evaluated each time a line of source code is executed.

Hardware breakpoint

For more information, see page 123.

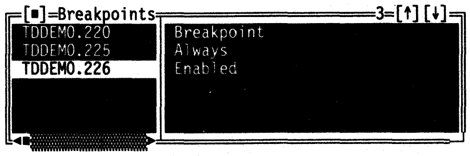
Use this command to access the Hardware Breakpoints Options dialog box. You must have the proper system setup in order to use hardware debugging.

Delete all

The Delete All command erases all the breakpoints you've set. Use this command when you want to start over from scratch.

The Breakpoints window

The Breakpoints window is accessed by choosing the View | Breakpoints command. This gives you a way of looking at and adjusting the conditions that trigger a breakpoint.



```
[■]=Breakpoints                    3=[↑][↓]
TDDEMO.220     Breakpoint
TDDEMO.225     Always
TDDEMO.226     Enabled
```

Figure 7.1
The Breakpoints window

The Breakpoints window has two panes; the Breakpoint List (left pane) shows a list of all the addresses at which breakpoints are set and the Breakpoint Detail (right pane) shows the details of the breakpoint highlighted in the left pane. Although a breakpoint can have several sets of actions and conditions associated with it, only the first set of details is displayed in the Breakpoint Detail pane.

The Breakpoints window has a local menu, which you access by pressing *Alt-F10*. If you have control-key shortcuts enabled, press *Ctrl* with the first letter of the command to access that command directly.

The Breakpoints window local menu

The commands in this menu let you add new breakpoints, delete existing breakpoints, and change how a breakpoint behaves.

```
Set options...
Add...
Remove
Delete all
Inspect
Group...
```

Set Options

Once a breakpoint is set, the Set Options command opens the Breakpoint Options dialog box, allowing you to modify the breakpoint. Using this box, you can

For a detailed explanation of the Breakpoint Options dialog box, see page 114.

- declare a global breakpoint
- disable/enable the breakpoint
- attach the breakpoint to a specific group
- access the Conditions and Actions dialog box

Add The Add command on the Breakpoints local menu opens the Breakpoint Options dialog box, much like the Set Options command does. The difference is that the cursor is positioned on an empty Address text box. Enter the address for which you'd like the breakpoint to be set into the Address text box. For example, if you'd like to set a breakpoint at line number 3201 in your Pascal source code, enter `MODNAME.3201` into the text box. If the line of code is in a module not displayed in the Module window, type the module name, followed by a period (.), and the line number. For example: `OTHERMOD.3201`.

The Add command can also be accessed by simply typing an address into the Breakpoint Window. After typing the first character of the address, the Breakpoint Options dialog box opens, placing you in the Address text box.

Once you've entered the breakpoint address, use the other commands in the Breakpoint Options dialog box to complete the breakpoint entry.

Remove The Remove command erases the currently highlighted breakpoint. *Del* is the hotkey for this command.

Delete all Delete All removes all breakpoints, both global and those set at specific addresses. You will have to set more breakpoints if you want your program to stop on a breakpoint. Use this command with caution!

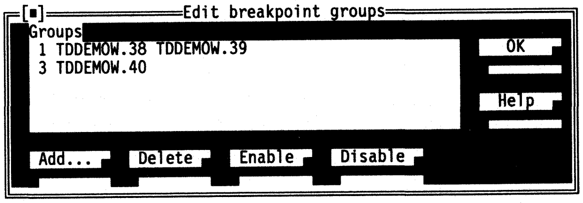
Inspect The Inspect command displays the source code line or assembler instruction that corresponds to the currently highlighted breakpoint item. If the breakpoint is set at an address that corresponds to a source line in your program, a Module window is opened and set to that line. Otherwise, a CPU window is opened, with the Code pane set to show the instruction at which the breakpoint is set.

You can also invoke this command by pressing *Enter* once you have the highlight bar positioned over a breakpoint.

Group The Group command allows you to gather breakpoints into groups. A breakpoint *group* is identified by a positive integer, generated automatically by TDW or assigned by you. The debugger automatically assigns a new group number to each breakpoint as it's created. The group number generated is the lowest number not already in use. Thus, if the numbers 1, 2, and 5 are already used by groups, the next breakpoint created is automatically given the group number 3.

Once a breakpoint is created, you may modify the breakpoint group number from the Breakpoint Options dialog box, placing the breakpoint into a group associated with other breakpoints. Grouping breakpoints together allows you to enable, disable, or remove a collection of breakpoints with a single action.

When the Group command is chosen from the Breakpoint window's local menu, the Edit Breakpoint Groups dialog box is displayed. This dialog box shows a listing of the current breakpoint groups and allows you to easily collect all functions within a module into a single group.



```
 1 TDDEMOW.38 TDDEMOW.39
 3 TDDEMOW.40
```

Figure 7.2 The Edit Breakpoint Groups dialog box

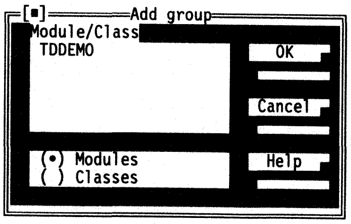
Groups

The Groups list box displays the currently assigned breakpoint groups.

Add

The Add button activates the Add Group dialog box.

Figure 7.3
The Add Group dialog box



The Add Group dialog box has a single list box and a single set of radio buttons that allow you to add all procedures in a single module, or all methods contained in an object, to a breakpoint group.

- The *Module/Class* list box displays a list of the modules or objects contained in the program loaded into the Module window. Highlight the desired module or object, then press OK to set breakpoints on all procedures or methods. All breakpoints set are collected into a single breakpoint group.
- Two radio buttons allow you to select what is displayed in the Module/Class list box:
  - The **M**odules radio button selects all modules contained in the current program, displaying them in the Module/Class list box.
  - The **C**lasses radio button selects all the Pascal objects contained in the current program for display in the Module/Class list box.

Delete

The Delete button in the Edit Breakpoint Groups dialog box removes the group currently highlighted in the Groups list box. All breakpoints in this group, along with their settings, will be erased.

Enable

The Enable button activates a breakpoint group that has been previously disabled.

Disable

The Disable command temporarily masks the breakpoint group that is currently highlighted in the Groups list box. Breakpoints that have been disabled are not erased; they are merely set aside for the current debugging session. Enabling the group reactivates all the settings for all the breakpoints in the group.

The Breakpoint Options dialog box

The Breakpoint Options dialog box is reached from the Breakpoints | At command, and from the Set Options and Add commands on the Breakpoints window local menu.

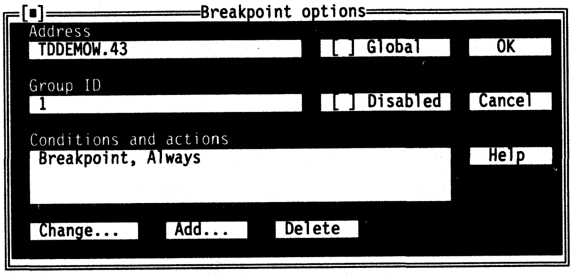


Figure 7.4
The Breakpoint Options dialog box

Address The Address text box contains the address tag associated with the currently highlighted breakpoint. Normally, you will not edit this field. However, if you want to change the name of the tag associated with the breakpoint, type the new name into the Address text box.

Group ID The Group ID text box allows you to assign the current breakpoint to a new or existing group. A breakpoint *group* is identified by a unique positive integer.

See page 112 for a description of breakpoint groups.

Global Global, when checked, enables *global checking*. This means that every time a source line is executed, the breakpoint conditions will be checked for validity. Because global breakpoints are tested after every line of code is executed, the Address field is set to `<not available>` since it is no longer pertinent.

For more information on global breakpoints, see page 121.

When you set a global breakpoint, you *must* set a condition that will trigger the global breakpoint. Otherwise, you'll end up with a breakpoint that activates on every line of source code (if this is the effect you want to achieve, use the Run | Trace Into command on the Main menu).

Disabled The Disabled check box turns off the current breakpoint. While this command is similar to the Toggle command on the Breakpoints menu (see page 109), Disable does not clear the breakpoint of its settings (as does the Toggle command). Disable simply masks the breakpoint until you want to reenable it by unchecking this box. When the breakpoint is reenabled, all settings previously made to the breakpoint will become effective.

This check box is useful if you have defined a complex breakpoint that you don't want to use just now, but will want to use again later. It saves you from having to delete the breakpoint, and then reenter it along with its complex conditions and actions.

Conditions and Actions The Conditions and Actions list box displays the set of conditions and actions associated with the current breakpoint.

Change The Change button, when selected, opens up the Conditions and Actions dialog box (see the next section). With this command, you can edit the item currently highlighted in the Conditions and Actions list box.

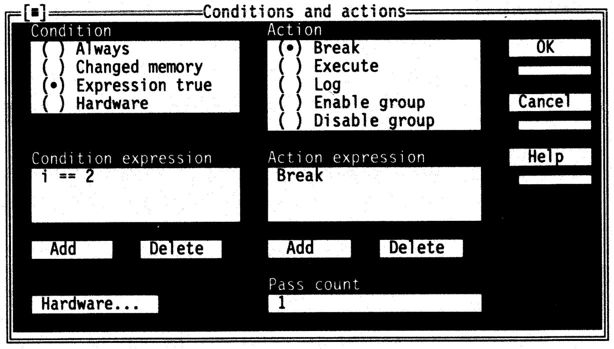
Add To add a new set of conditions and actions to the current breakpoint, select Add. Like the Change command above, Add opens the Conditions and Actions dialog box.

Delete The Delete command removes the currently highlighted item in the Condition and Actions list box from the breakpoint definition.

The Conditions and Actions dialog box

When you choose either the Change or the Add button from the Breakpoint Options dialog box, you're presented with the Conditions and Actions dialog box.

Figure 7.5
The Conditions and Actions dialog box



```
[■]══════════════Conditions and actions══════════════
 Condition                  Action
 ( ) Always                 (•) Break               OK
 ( ) Changed memory         ( ) Execute
 (•) Expression true        ( ) Log                 Cancel
 ( ) Hardware               ( ) Enable group
                            ( ) Disable group
 Condition expression       Action expression       Help
 i == 2                     Break


 Add        Delete          Add        Delete

                            Pass count
 Hardware...                1
```

See "Customizing breakpoints" on page 120 for details on modifying breakpoints.

When a breakpoint is set on a line of source code, its default characteristics are Always Break execution when the line of code is encountered. With the Conditions and Actions dialog box, you can customize the conditions under which the breakpoint will be activated and specify different actions that take place once the breakpoint does trigger.

You'll customize breakpoints through two sets of radio buttons and three text entry boxes. In addition, for global breakpoints, a Hardware button leads to the Hardware Breakpoints Options dialog box, allowing you to specify hardware breakpoint conditions.

The condition radio buttons

The Condition radio buttons have four settings:

Always

When Always is chosen, it indicates that no additional conditions need be true for the breakpoint to trigger; it will be triggered each time program execution encounters the breakpoint.

Changed memory

See page 121 for more information on Changed memory breakpoints.

A Changed Memory breakpoint watches a memory variable or object; the breakpoint is triggered if the object changes value. Use the Condition Expression input box to enter an expression representing the data object you want to watch.

Expression true

See page 122 for details on expressions.

The Expression True button allows the breakpoint to be triggered when an expression becomes true (nonzero). Use the Condition Expression input box to enter an expression that's evaluated each time the breakpoint is encountered.

Hardware

See page 123 for more information about hardware breakpoints.

Causes the breakpoint to be triggered by the hardware-assisted device driver. Because you can use hardware assistance only with a global breakpoint, you must check the Global check box in the Breakpoint Options dialog box before you can access this option.

The Hardware I Breakpoint command offers an easy way to set hardware breakpoints.

You must select the Hardware radio button before the Hardware button at the bottom of the dialog box can become active. Pushing that button displays the Hardware Breakpoint Options dialog box. The choices you can make in this box are described in the online text file HDWDEBUG.TD.

The action radio buttons

The Action radio buttons have five settings:

Break

Break causes your program to stop when the breakpoint is triggered. The TDW screen reappears, and you can once again enter commands to look around at your program's data structures.

Execute

Execute causes an expression to be executed. Enter the expression in the Action Expression input box. The expression should have some side effect, such as setting a variable to a value. By executing an expression that has side effects each time a breakpoint is triggered, you can effectively "splice in" new pieces of code before a given source line. This is useful when you want to alter the behavior of a routine to test a diagnosis or bug fix. This saves you from going through the compile-and-link cycle just to test a minor change to a routine.

Of course, this technique is limited to the insertion of an expression before an already existing line of code is executed; you can't use this technique to modify existing source lines directly.

Log

See Chapter 9 for a description of expressions and side effects.

The Log button causes the value of an expression to be recorded in the Log window. You are prompted for the expression whose value you want to log. Be careful that the expression doesn't have any unexpected side effects.

Enable group

The Enable Group action button allows for a breakpoint to reactivate a group of breakpoints that have been previously disabled.

Disable group

The Disable Group radio button lets you disable a group of breakpoints. When a group of breakpoints is disabled, the breakpoints are not erased, they are simply masked for the debugging session.

Setting conditions and actions

The most important step when setting up breakpoints is specifying the conditions under which the breakpoint triggers and specifying the actions to be taken once the breakpoint takes effect. Two text boxes control these settings, the Condition Expression text box and the Action Expression text box.

Condition expression

When you choose either a Changed Memory, Expression True, or Hardware Condition radio button, you must provide a set of conditions so TDW knows when to trigger the breakpoint. A *condition set* consists of one or more expressions; each condition has to evaluate true in order for the whole set to evaluate true.

For more information on specifying breakpoint actions, see the "Action Expression" section that follows.

A condition set is associated with a set of actions. When the condition set evaluates true, the corresponding action set is performed.

To add a condition set to a breakpoint,

1. Select either the Changed Memory, Expression True, or Hardware radio button.
2. Select the Add button located under the Condition Expression text box.
3. Enter the condition or variable expression into the Condition Expression text box.

4. If you want more than one variable or condition to be tested for a particular action set, repeat steps 2 and 3 until all expressions have been added to the Condition Expression text box.
5. Once you've specified a condition set, use the Action Expression text box to list the action(s) you'd like to take when the breakpoint triggers.

A single breakpoint may have several condition and action sets associated with it. If you want more than one set of conditions and actions assigned to a single breakpoint, choose OK after you have entered the first series of conditions and actions. This will close the Conditions and Actions dialog box and return you to the Breakpoint Options dialog box. From here, choose Add to enter a new set of conditions and actions. When a breakpoint has multiple condition and action sets, each one will be evaluated in the order that they were entered. If more than one action set evaluates to true, then more than one set of actions will be performed.

The Delete button located below the Condition Expression text box lets you remove the currently highlighted expression from the Condition Expression text box. Select this button if you want to delete a condition from the condition set.

Action expression

When either an Execute, Log, Enable Group, or Disable Group Action radio button is chosen, an action set must be provided so TDW knows what to do when the breakpoint triggers. An *action set* is composed of one or more actions.

To add an action set to a breakpoint,

1. Select either the Execute, Log, Enable Group, or Disable Group radio button.
2. Select the Add button located under the Action Expression text box.
3. Enter the action into the Action Expression text box.

   To perform more than one action when the breakpoint triggers, repeat steps 2 and 3 until all actions have been added to the Action Expression text box.
4. When you have finished entering actions, choose OK from the Conditions and Actions dialog box.

☞ If the Enable Group or Disable Group radio button is chosen, simply type the breakpoint group number into the Action Expression text box to reference the group that you want to enable or disable.

The Delete button located below the Action Expression text box lets you remove the currently highlighted action from the action set.

Pass count The Pass Count input box lets you set the number of times the breakpoint condition set must be met before the breakpoint is triggered. The default number is 1. The pass count is decremented only when the entire condition set attached to the breakpoint is true. This means that if you set a pass count to *n*, the breakpoint is triggered the *n*th time that the condition set is true.

Customizing breakpoints

In addition to simply stopping your program at a particular point, greater control can be given to debugging by stipulating when a breakpoint should take action, and what it should do when it triggers.

Simple breakpoints

When a breakpoint is initially set, it is given the default setting of Always Break. Once a simple breakpoint is set, the actions and conditions of the breakpoint may be customized. There are a number of ways to set a simple breakpoint, each one being convenient in different circumstances:

- Move to the desired source line in a Module window (or Code pane of a CPU window) and issue the Breakpoints | Toggle command (or press *F2*, or click either of the first two characters of the line with your mouse). Doing this on a line that already has a breakpoint set causes that breakpoint to be deleted.
- Issue the Add local menu command from the Breakpoint List pane of the Breakpoints window and enter a code address at which to set a breakpoint. (A code address has the same format as a pointer in the language you're using. See Chapter 9 about expressions.)

- Issue the Breakpoints | At command to set a breakpoint at the current line in the Module window.

Global breakpoints

When a breakpoint is made global, TDW will check the breakpoint on the execution of every line of code. If the set of conditions evaluates true, then the corresponding set of actions will be executed.

☞ If you want a global check to occur on every machine code instruction, set a global breakpoint, and press *F9* from within the CPU window. This type of code monitoring should only be done once you have isolated a small area of your program known to contain a problem. The CPU window can then be used to locate the exact position of the difficulty.

You must check Global if you want to set hardware breakpoints.

Since a debugger action will occur on every line of source code or machine instruction, global breakpoints greatly slow the execution of your program. Be careful with your use of global breakpoints; they should be used only if you want to find out exactly when a variable changes value, when some condition becomes true, or when your program is "bashing" data.

The Breakpoints menu offers shortcuts for defining global breakpoints. For more information on the Changed Memory Global and Expression True Global commands, see page 109.

Often, global breakpoints are used to watch for when a data item changes value. In this situation, TDW checks the area of memory for change after the execution of every line of code. As an alternative to a global breakpoint, you may want to specify a breakpoint that only watches for a change when a specific source statement is reached. This is a lot more efficient, since it reduces the amount of processing TDW does in order to detect the change (in this case, TDW isn't concerned with when the item has changed, only that it has changed).

Changed memory breakpoints

When you want to find out where in your program a certain data object is being changed, first set a breakpoint using one of the techniques outlined in the preceding section. Then, using the Changed Memory radio button in the Conditions and Actions dialog box, enter an expression that refers to the memory area you want to watch along with an optional count of the number of objects to track. The total number of bytes in the watched area is the size of the object that the expression references times the number of objects. For example, suppose you have declared the following Pascal array:

```
Temperature: array[1..50] of Integer;
```

If you want to watch for a change in the first ten elements of this array, enter the following item into the Condition Expression input box:

```
Temperature[1], 10
```

The area watched is 20 bytes long, since an **Integer** is 2 bytes and you said to watch 10 of them.

If the Changed Memory breakpoint is global, your program executes much more slowly because the memory area is checked for change after every source line has been executed. If you've installed a hardware device driver, TDW will try to set a hardware breakpoint to watch for a change in the data area. Different hardware debuggers support different numbers and types of hardware breakpoints. You can see if a breakpoint is using the hardware by opening a Breakpoint window with the View | Breakpoints command. Any breakpoint that is hardware assisted will have an asterisk (*) beside it. These breakpoints are much faster than global breakpoints that are not hardware assisted.

Conditional expressions

There are many occasions when you won't want a breakpoint to be triggered every time a certain source statement is executed, particularly if that line of code is executed many times before the occurrence you are interested in. TDW gives you two ways to qualify when a breakpoint is actually triggered: *pass counts* and *conditions*.

Scope of breakpoint expressions

See Chapter 9 for a complete discussion of scopes and scope overrides.

Both the action that a breakpoint performs and the condition under which it is triggered can be controlled by an expression you supply. That expression is evaluated using the scope of the address at which the breakpoint is set, not the scope of the current location where the program is stopped. This means that your breakpoint expression can use only variable names that are valid at the address in your program where you set the breakpoint, unless you use scope overrides.

If you want to set a breakpoint for an expression in a module that isn't currently loaded and TDW cannot find that expression, you can use either a scope override to specify the file that contains the expression or the View | Module command to change modules.

If you use variables that are local to a routine as part of an expression, that breakpoint will execute much more slowly than a breakpoint that uses only global or module local variables.

Hardware breakpoints

See page 12 for information on setting up device drivers for hardware debugging.

A hardware breakpoint uses hardware debugging support, either through a hardware debugging board or through the debugging registers of the Intel 80386 (or higher) processor. If your system is set up for hardware debugging (File | Get Info shows Breakpoints set to Hardware), you can set a hardware breakpoint using one of the following methods:

- Choose Breakpoints | Changed Memory Global, the most common use of hardware breakpoints.
- Choose Breakpoints | Hardware.
- Display the Breakpoint Options menu (choose Breakpoints | At or the Set Options command of the View | Breakpoints window local menu), then do the following:

  1. Check the Global check box.
  2. Push the Change button.
  3. In the Conditions and Actions dialog box, choose the Hardware radio button to turn on the Hardware pushbutton at the bottom of the dialog box.
  4. Push the Hardware pushbutton to display the Hardware Breakpoint Options dialog box.
  5. Choose the options you want from this dialog box. The options are described in the online text file HDWDEBUG.TD.

Logging variable values

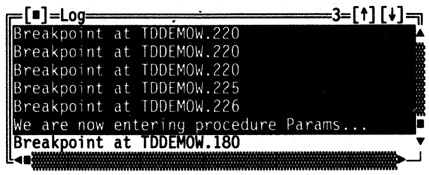
Sometimes, you may find it useful to log the value of certain variables each time you reach a certain place in your program. You can log the value of any expression, including, for example, the values of the parameters that a procedure is called with. By looking at the log each time the procedure is called, you can determine when it was called with erroneous parameters.

Be careful of side effects when logging expressions.

Choose the Log radio button from the Breakpoint Options dialog box. You are prompted for the expression whose value is to be logged each time the breakpoint is triggered.

The Log window

You create a Log window by choosing the View | Log command. This window lets you review a list of significant events that have taken place in your debugging session.



```
Breakpoint at TDDEMOW.220
Breakpoint at TDDEMOW.220
Breakpoint at TDDEMOW.220
Breakpoint at TDDEMOW.225
Breakpoint at TDDEMOW.226
We are now entering procedure Params...
Breakpoint at TDDEMOW.180
```

Figure 7.6
The Log window

The Log window shows a scrolling list of the lines output to the window. If more than 50 lines have been written to the log, the oldest lines are lost from the top of the scrolled list. If you want to change the number of lines in the list, use the TDWINST customization program (described in the online text file UTILS.TDW). You can also preserve the entire log, continuously writing it to a disk file, by using the Open Log File local menu command.

Here's a list of what can cause lines to be written to the log:

- Your program stops at a location you specified. The location it stops at is recorded in the log.
- You issue the Add Comment local menu command. You are prompted for a comment to write to the log.
- A breakpoint is triggered that logs the value of an expression. This value is put in the log.
- You use the Edit | Dump Pane to Log command (from the menu bar) to record the current contents of a pane in a window.
- You are debugging a Windows application and use the Display Windows Info command on the Log window local menu to write global heap information, local heap information, or the module list to the log.

- You are debugging a Windows application, have used the View | Windows Messages command to display the Windows Messages window, and are now in the local menu of the Messages pane of that window. You toggle Send to Log Window to *Yes* so all messages coming to this window will also go to the Log window.

The Log window local menu

```
Open log file...
Close log file
Logging     Yes
Add comment...
Erase log
Display Windows info...
```

The commands in this menu let you control writing the log to a disk file, stopping and starting logging, adding a comment to the log, clearing the log, and writing information about a Windows program to the log.

Alt-F10 pops up the Log window local menu. If you have control-key shortcuts enabled, pressing *Ctrl* and the first letter of the command accesses the command directly.

Open Log File

Causes all lines written to the log to be written to a disk file as well. A dialog box appears that prompts you for the name of the file to write the log to (or you can select a directory and file from the list boxes).

When you open a log file, all the lines already displayed in the log window's scrolling list are written to the disk file. This lets you open a disk log file *after* you see something interesting in the log that you want to record to disk.

If you want to start a disk log that does not start with the lines already in the Log window, first choose Erase Log before choosing Open Log File.

Close Log File

Stops writing lines to the log file specified in the Open Log File local menu command, and closes the file.

Logging

Enables or disables the log, controlling whether anything is actually written to the Log window.

Add Comment

Lets you insert a comment in the log. You are prompted for a line of text that can contain any characters you want.

Erase Log Clears the log list. The Log window will now be blank. Only the log in memory is affected, not the parts of the log that have been written to a disk file.

Display Windows Info Displays the Windows Information dialog box, which lets you list global heap information, local heap information, or the list of modules making up your application. See page 166 in Chapter 11 for an explanation of how to use this feature.

C H A P T E R

8

Examining files

TDW treats disk files as a natural extension of the program you're debugging. You can examine any file on the disk, viewing it either as ASCII text or as hex data.

This chapter shows you how to examine disk files that contain your program source code and other files on disk.

Examining program source files

Loading and debugging Windows DLL modules is described in Chapter 11 on page 169.

Program source files are your source files that are compiled to generate an object module (an .EXE file). You usually examine them when you want to look at the behavior or design of a portion of your code. During debugging, you often need to look at the source code for a routine to verify either that its arguments are valid or that it is returning a correct value.

As you step through your program, TDW automatically displays the source code for the current location in your program.

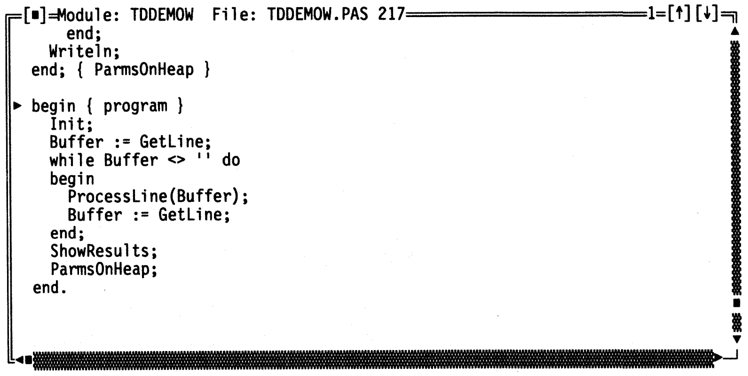
☞ Files that are included in a source file by a compiler directive and that generate line numbers are also considered to be program source files, even though they don't appear in the Pick a Module list pane when you choose View | Module. To select one of these files, you must use the local menu File command.

You should always use a Module window to look at your program source files because doing so informs TDW that the file is a source module. TDW then lets you do things like setting breakpoints or examining program variables simply by moving to the appropriate place in your file. These techniques and others are described in the following sections.

The Module window

Before you can open a module window, you must have a program loaded. You create a Module window by choosing the View | Module command from the menu bar (or pressing the hot key, *F3*).

Figure 8.1
The Module window



```
[■]=Module: TDDEMOW  File: TDDEMOW.PAS 217=========================1=[↑][↓]=
     end;
   Writeln;
 end; { ParmsOnHeap }

► begin { program }
   Init;
   Buffer := GetLine;
   while Buffer <> '' do
   begin
     ProcessLine(Buffer);
     Buffer := GetLine;
   end;
   ShowResults;
   ParmsOnHeap;
 end.
```

See page 170 for a description of this Dialog box.

A dialog box appears in which you can enter the name of the module or DLL you want to view.

When you run TDW, you need both the .EXE file and the original source file.

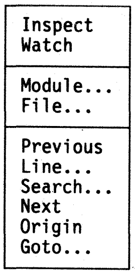
TDW then loads the source file for the module you select. If you select a source module (and not a DLL), TDW searches for the source file in the following places:

1. in the directory where the compiler found the source file
2. in the directories specified by the Options | Path for Source command or the **–sd** command-line option
3. in the current directory
4. in the directory that contains the program you're debugging

Module windows show the contents of the source file for the module you've selected. The title of the Module window shows the name of the module you're viewing, along with the source file name and the line number the cursor is on. An arrow (►) in the first column of the window shows the current program location (CS:IP).

If the word *modified* appears after the file name in the title, the file has been changed since it was last compiled or linked to make the program you are debugging. In this case, the routines in the updated source file may no longer have the same line numbers as those in the version used to build the program you are debugging. If the line numbers are different, the arrow that shows the current program location (CS:IP) will be displayed on the wrong line.

The Module window local menu



The Module window local menu provides a number of commands that let you move around in the displayed module, point at data items and examine them, and set the window to display a new file or module.

You will probably use this menu more than any other menu in TDW, so you should become quite familiar with its various options.

Use the *Alt-F10* key combination to pop up the Module window local menu. If you have control-key shortcuts enabled, you can access local menu commands without popping up the menu: Use the *Ctrl* key with the highlighted letter of a command to access that command (for example, *Ctrl-S* for Search).

Inspect

Opens an Inspector window to show you the contents of the program variable at the current cursor position. If the cursor isn't currently on a variable, you're prompted to enter one.

Because this command saves you from having to type in each name you are interested in, you'll end up using it a lot to examine the contents of your program variables.

Watch

If the cursor isn't currently on a variable, you're prompted to enter one.

Adds the variable at the current cursor position to the Watches window. Putting a variable in the Watches window lets you monitor the value of that variable as your program executes.

Module

Lets you view a different module by picking the one you want from the list of modules displayed. This command is useful when you are no longer interested in the current module, and you don't want to end up with more Module windows onscreen.

File Lets you switch to view one of the other source files that makes up the module you are viewing. Pick the file that you want to view from the list of files presented. Most modules only have a single source file that contains code. Other files included in a module usually only define constants and data structures. Use this command if your module has source code in more than one file.

Use View | Module to look at the first file. If you want to see more than one, use View | Another | Module to open subsequent Module windows.

Previous Returns you to the last source module location you were viewing. You can also use this command to return to your previous location after you've issued a command that changed your position in the current module.

Line Positions you at a new line number in the file. Enter the new line number to go to. If you enter a line number after the last line in the file, you will be positioned at the end of the file.

Search Searches for a character string, starting at the current cursor position. Enter the string to search for. If the cursor is positioned over something that looks like a variable name, the Search dialog box will come up initialized to that name. Also, if you have marked a block in the file using the *Ins* key, that block will be used to initialize the Search dialog box. This saves you from typing if what you want to search for is a string that is already in the file you are viewing.

You can search using simple wildcards, with ? indicating a match on any single character, and * matching zero or more characters. The search does not wrap around from the end of the file to the beginning. To search the entire file, go to the first line by pressing *Ctrl-PgUp*.

Next Searches for the next instance of the character string you specified with the Search command; you can only use this command after initially choosing Search.

Sometimes, Search matches an unexpected string before reaching the one you really wanted to find. Next lets you repeat the search without having to reenter what you want to search for.

Origin Positions you at the module and line number that is the current program location (CS:IP). If the module you are currently viewing is not the module that contains the current program location, the Module window will be switched to show that module. This command is useful after you have looked around in your code and want to return to where your program is currently stopped.

Goto Positions you at any location within your program. Enter the address you want to examine; you can enter a procedure name or a hex address. See Chapter 9 for a complete description of the ways to enter an address.

If the address doesn't have a corresponding source line, a CPU window is opened.

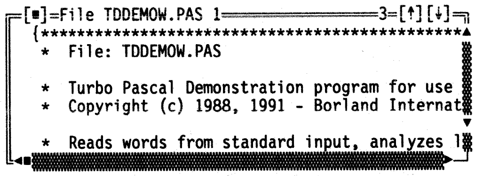
You can also invoke this command by simply starting to type the label to go to. This brings up a dialog box exactly as if you had chosen the Goto command. Entering the label name is a handy way to invoke this frequently used command.

Examining other disk files

You can examine any file on your system by using a File window. You can view the file either as ASCII text or as hex data bytes, using the Display As command described in a later section of this chapter.

The File window

You create a File window by choosing View | File from the menu bar. You can use DOS-style wildcards to get a list of file choices, or you can type a specific file name to load.

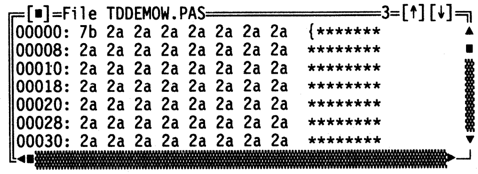


```
{*********************************************
 *  File: TDDEMOW.PAS

 *  Turbo Pascal Demonstration program for use
 *  Copyright (c) 1988, 1991 - Borland Internat

 *  Reads words from standard input, analyzes 1
```

Figure 8.2
The File window

File windows show the contents of the file you've selected. The name of the file you are viewing is displayed at the top of the window, along with the line number the cursor is on if the file is displayed as ASCII text.

When you first create a File window, the file appears either as ASCII text or as hexadecimal bytes, depending on whether the file contains what TDW thinks is ASCII text or binary data. You can switch between ASCII and hex display at any time using the Display As local menu command described later.



```
00000: 7b 2a 2a 2a 2a 2a 2a 2a  {*******
00008: 2a 2a 2a 2a 2a 2a 2a 2a  ********
00010: 2a 2a 2a 2a 2a 2a 2a 2a  ********
00018: 2a 2a 2a 2a 2a 2a 2a 2a  ********
00020: 2a 2a 2a 2a 2a 2a 2a 2a  ********
00028: 2a 2a 2a 2a 2a 2a 2a 2a  ********
00030: 2a 2a 2a 2a 2a 2a 2a 2a  ********
```

Figure 8.3
The File window showing hex data

The File window local menu

The File window local menu has a number of commands for moving around in a disk file, changing the way the contents of the file are displayed, and making changes to the file.

```
Goto...
Search...
Next
-----------------
Display as   Ascii
File...
```

Use the *Alt-F10* key combination to pop up the File window local menu or, if you have control-key shortcuts enabled, use the *Ctrl* key with the highlighted letter of the desired command to access the command without invoking the local menu.

Goto Positions you at a new line number or offset in the file. If you are viewing the file as ASCII text, enter the new line number to go to. If you are viewing the file as hexadecimal bytes, enter the offset from the start of the file at which to start displaying. You can use the full expression parser for entering the offset. If you enter a line number after the last line in the file or an offset beyond the end of the file, TDW positions you at the end of the file.

Search Searches for a character string, starting at the current cursor position. You are prompted to enter the string to search for. If the cursor is positioned on something that looks like a symbol name, the Search dialog box comes up initialized to that name. Also, if you have marked a block in the file using the *Ins* key, that block will be used to initialize the Search dialog box. This saves you from typing if what you want to search for is a string that is already in the file you are viewing. The format of the search string depends on whether the file is displayed in ASCII or hex.

If the file is displayed in ASCII, you can use simple DOS wildcards, with ? indicating a match on any single character, and * matching 0 or more characters.

See page 142 for complete information about byte lists.

If the file is displayed in hexadecimal bytes, enter a byte list consisting of a series of byte values or quoted character strings, using the syntax of whatever language you are using for expressions.

For example, if the language is C++, a byte list consisting of the hex numbers 0408 would be entered as follows:

```
0x0804
```

If the language is Pascal, the same byte list is entered as

```
$0804
```

The search does not wrap around from the end of the file to the beginning. To search the entire file, go to the first line of the file by pressing *Ctrl-PgUp.*

You can also invoke this command by simply starting to type the string that you want to search for. This brings up a dialog box exactly as if you had specified the Search command.

Next Searches for the next instance of the character string you specified with the Search command; you can only use this command after initially choosing Search.

Next is useful when your Search command didn't find the instance of the string you wanted; you can keep issuing this command until you find what you want.

Display As Toggles between displaying the file as ASCII text or as hexadecimal bytes.

- If you choose ASCII display, the file appears as you are used to seeing it on the screen in an editor or word processor.
- If you choose Hex display, each line starts with the hex offset from the beginning of the file for the bytes on the line. Eight bytes of data are displayed on a line. To the right of the hex display of the bytes, the display character for each byte appears. The full display character set can be displayed, so byte values less than 32 or greater than 127 appear as the corresponding display symbol.

File Lets you switch to a different file. You can use DOS wildcards to get a list of file choices, or you can type a specific file name to load. File lets you view a different file without putting a new File window onscreen. If you want to view two different files or two parts of the same file simultaneously, choose View | Another | File to make another File window.

C H A P T E R

9

Expressions

Expressions can be a mixture of symbols from your program (that is, variables and names of routines), and constants and operators from one of the supported languages: C, Pascal, or assembler.

Each language evaluates an expression differently.

TDW can evaluate expressions and tell you their values. You can also use expressions to indicate data items in memory whose value you want to know. You can supply an expression in any dialog box that asks for a value or an address in memory.

Use Data | Evaluate/Modify to open the Evaluate/Modify dialog box, which tells you the value of an expression. (You can also use this dialog box or the Watches window as a simple calculator.)

In this chapter, you'll learn how TDW chooses which language to use for evaluating an expression and how you can make it use a specific language. We describe the components of expressions that are common to all the languages, such as source line numbers and access to the processor registers. We then describe the components that can make up an expression in each language, including constants, program variables, strings, and operators. For each language, we also list the operators that TDW supports and the syntax of expressions.

For a complete discussion of Pascal and assembler expressions, refer to your *Turbo Pascal for Windows Language Guide.*

Choosing the language for expression evaluation

TDW normally determines which expression evaluator and language to use from the language of the current module. This is the module in which your program is stopped. You can override this by using the Options | Language command to open the Expression Language dialog box; in it you can set radio buttons to Source, Pascal, C, or Assembler. If you choose Source, expressions are evaluated in the manner of the module's language. (If TDW can't determine the module's language, it uses the expression rules for inline assembler.)

Usually, you let TDW choose which language to use. Sometimes, however, you'll find it useful to set the language explicitly; for example, when you are debugging an assembler module that is called from one of the other languages. By explicitly setting expression evaluation to use a particular language, you can access your data in the way you refer to it with that language, even though your current module uses a different language.

Sometimes it's convenient to treat expressions or variables as if they had been written in a different language; for example, if you're debugging a Pascal program, assembly language conventions might offer an easier way to change the value of a byte stored in a string.

If your initial choice of language is correct when you enter TDW, you should have no difficulty using other language conventions. TDW still retains information about the original source language and handles the conversions and data storage appropriately. If the language seems ambiguous, TDW defaults to assembly language.

Even if you deliberately choose the wrong language when you enter TDW, it will still be able to get some information about the original source language from the symbol table and the original source file. Under some circumstances, however, it may be possible to cause TDW to store data incorrectly.

Code addresses, data addresses, and line numbers

Normally, when you want to access a variable or the name of a routine in your program, you simply type its name. However, you can also type an expression that evaluates to a memory pointer, or specify code addresses as source line numbers by preceding the line number with a number sign (#), like #123 (C, C++, and Assembler only). The next section describes how to access symbols outside the current scope.

Of course, you can also specify a regular segment:offset address, using the hexadecimal syntax for the source code language of your program:

Language	Format	Example
Pascal	$*nnnn*	$1234:$0010
Assembler	*nnnn*h	1234h:0010h 1234h:0B234h

In assembler, hex numbers starting with A to F must be prefixed with a zero.

Accessing symbols outside the current scope

Where the debugger looks for a symbol is known as the *scope* of that symbol. Accessing symbols outside of the current scope is an advanced concept that you don't really need to understand in order to use TDW in most situations.

Normally, TDW looks for a symbol in an expression the same way a compiler would. For example, Pascal first looks in the current procedure or function, then in an "outer" subprogram (if the active scope is nested inside another), then in the implementation section of the current unit (if the current scope resides in a unit), and then for a global symbol.

For example, in the Watches window, you could enter different line numbers for the variable *nlines* so you could see how its value changes in different routines in the current module. To watch the variable both on line 51 and on line 72, you would make the following entries in the Watches window:

```
#51#nlines
#72#nlines
```

Here are some examples of valid symbol expressions with scope overrides. There is one example for each of the legal combinations of elements that you can use to override a scope.

The first six examples show various ways of using line numbers to generate addresses and override scopes:

`#123`
: Line 123 in the current module

`#123#myvar1`
: Symbol *myvar1* accessible from line 123 of the current module

`#mymodule#123`
: Line 123 in module *mymodule*

`#mymodule#123#myvar1`
: Symbol *myvar1* accessible from line 123 in module *mymodule*

`#mymodule#file1.cpp#123`
: Line 123 in source file *file1.cpp,* which is part of module *mymodule*

`#mymodule#file1.cpp#123#myvar1`
: Symbol *myvar1* accessible from line 123 in source file *file1.cpp,* which is part of *mymodule*

The next six examples show various ways of overriding the scope of a variable by using a module, file, or function name:

`#myvar2`
: Same as *myvar2* without the #

`myfunc#myvar2`
: Variable *myvar2* accessible from routine *myfunc*

`#mymodule#myvar2`
: Variable *myvar2* accessible from module *mymodule*

`#mymodule#myfunc#myvar2`
: Variable *myvar2* accessible from routine *myfunc* in module *mymodule*

`#mymodule#file2.c#myvar2`
: Variable *myvar2* accessible from *file2.c,* which is included in *mymodule*

`#mymodule#file2.c#myfunc`
: *myfunc* defined in file *file2.c,* which is included in *mymodule*

OOP

The following four examples show how to use scope override syntax with C++ classes, objects, member functions, and data members:

`AnObject#AMemberVar`

Data member *AMemberVar* accessible in object *AnObject* accessible in the current scope

`AnObject#AMemberF`

Member function *AMemberF* accessible in object *AnObject* accessible in the current scope

`#AModule#AnObject#AMemberVar`

Data member *AMemberVar* accessible in object *AnObject* accessible in module *AModule*

`#AModule#AnObject#AClass::AMemberVar`

Data member *AMemberVar* of class *AClass* accessible in object *AnObject* accessible in module *AModule*

Scope override tips

The following tips might help you when overriding scope in C, C++, and Turbo Assembler programs:

1. If you use a file name in a scope override statement, it must be preceded by a module name.
2. If a file name has an extension, such as .ASM, .C, or .CPP, you must specify the extension; Turbo Debugger doesn't try to determine the extension itself.
3. If a function name is the first item in a scope override statement, it must not have a # in front of it. If there's a #, Turbo Debugger interprets the function name as a module name.
4. Any variable you access through scope override syntax must have been initialized already. An automatic variable doesn't have to be in scope, but its function must have run already.
5. If you're trying to access an automatic variable that's no longer in scope, you must use its function name as part of the scope override statement.

- The scope of a template depends on the current location in the program. Watches and Inspector windows on template expressions are dependent on the current object the program is in.
- A nested class is in the scope of the class it's nested in. The scope of a nested class isn't global to the program.

Overriding scope in Pascal programs

Use a period (.) to separate the components of the scope.

The following syntax describes scope overriding; brackets ([]) indicate optional items:

```
[unit.][procedurename.]variablename
```

or

OOP

```
[unit.][[objecttype.]|[objectinstance.]][method.]fieldname
```

If you don't specify a unit, the current unit is assumed.

Here are some examples of valid symbol expressions with scope overrides. There is one example for each of the legal combinations of elements that you can use to override a scope.

These examples show various ways of overriding the scope of a variable by using a module or procedure name:

`MyVar2`
: Variable *MyVar2* in the current scope

`MyProc.MyVar2`
: Variable *MyVar2* accessible from routine *MyProc*

`MyUnit.MyVar2`
: Variable *MyVar2* accessible from unit *MyUnit*

`MyUnit.MyProc.MyVar2`
: Variable *MyVar2* accessible from routine *MyProc* in unit *MyUnit*

OOP

The following examples show how to use scope override syntax with object types, object instances, fields, and methods:

`AnInstance`
: Instance *AnInstance* accessible in the current scope.

`AnInstance.AField`
: Field *AField* accessible in instance *AnInstance* accessible in the current scope

`AnObjectType.AMethod`
: Method *AMethod* accessible in object type *AnObjectType* accessible in the current scope

`AnInstance.AMethod`
: Method *AMethod* accessible in instance *AnInstance* accessible in the current scope

`AUnit.AnInstance.AField`
: Field *AField* accessible in instance *AnInstance* accessible in unit *AUnit*

`AUnit.AnObjectType.AMethod`
: Method *AMethod* accessible in object type *AnObjectType* accessible in unit *AUnit*

`AUnit.AnInstance.AMethod.ANestedProc.AVar`
: Local variable *AVar* accessible in nested procedure *ANestedProc* accessible in method *AMethod* accessible in instance *AnInstance* accessible in unit *AUnit*

Scope override tips

The following tips might help you when overriding scope in Pascal programs:

1. Any variable you access through scope override syntax must have been initialized already. The procedure or function containing a local variable doesn't have to be in scope, but it must have run already.
2. If you are trying to access a local variable that's no longer in scope, you must use its procedure or function name as part of the scope override statement.
3. You can't use a line number or a file name as part of a Pascal scope override statement. However, you can use Options | Language to change the language to C so you *can* use line number syntax.

Scope and DLLs

Your .EXE and .DLL files must all be in the same directory.

Because TDW simultaneously loads the symbol tables of the current module of your .EXE file and of any DLLs it accesses that have source code and symbol tables, you might not have immediate access to variables in your DLLs from your .EXE module (or to the variables in your .EXE if you're currently in a DLL).

TDW looks for a variable first in the symbol table of the current module or DLL, and then in any other symbol tables in order of loading. If a variable has the same name in multiple DLLs or in your .EXE and one or more DLLs, TDW sees only the first instance it finds. You can't use scope override syntax to access any such variables; instead, you must press *F3* and use the Load Modules and DLLs dialog box to load the appropriate module or DLL.

TDW loads symbol tables for the following items:

1. the current module of your .EXE file
2. any DLL you explicitly load using the Symbol Load command in the Load Modules and DLLs dialog box (displayed with *F3* or View | Module)
3. any DLL you step into from your program

Implied scope for expression evaluation

Whenever TDW evaluates an expression, it must decide where the *current scope* is for any symbol names without an explicit scope override. Determining scope is important because in many languages you can have symbols inside functions or procedures with the same name as global symbols, and TDW must know which instance of a symbol you mean.

TDW usually uses the current cursor position as the context for determining the scope. Thus, you can set the scope where an expression will be evaluated by moving the cursor to a specific line in a Module window.

One result is that if you've moved the cursor off the current line where your program is stopped, you might get unexpected results from evaluating expressions. If you want to be sure that expressions are evaluated in your program's current scope, use the Origin local menu command in the Module window to return to the current location in the source code. You can also set the expression scope by moving around inside the Code pane of a CPU window, by moving the cursor to a routine in the Stack window, or by moving the cursor to a routine name in a Variables window.

Byte lists

Several commands ask you to enter a list of bytes, including the Search and Change local menu commands in the Data pane of the CPU window, and the Search local menu command of the File window when it's displaying a file in hexadecimal format.

A *byte list* can be any mixture of scalar (non-floating-point) numbers and strings in the syntax of the current language, determined by the Options | Language command. Both strings and scalars use the same syntax as expressions. Scalars are converted into a corresponding byte sequence. For example, a Longint value of *123456* becomes a 4-byte hex quantity *40 E2 01 00*.

Language	Byte list	Hex data
Pascal	'ab'$04'c'	61 62 04 63
Assembler	1234 "AB"	34 12 41 42
C	"ab" 0x04 "c"	61 62 04 63

Pascal expressions

TDW supports the Pascal expression syntax, with the exception of string concatenation and set operators. A Pascal expression consists of a mixture of symbols, operators, strings, variables, and constants. The following sections describe each of the components that make up an expression.

Symbols

Symbols in Pascal are user-defined names for data items or routines in your program. A Pascal symbol name can start with a letter (*a-z*, *A-Z*) or an underscore (_). Subsequent characters in the name can contain the digits (*0* to *9*) and the underscore, as well as letters.

Normally, a symbol obeys the Pascal scoping rules, with "nested" local symbols overriding other symbols of the same name. You can override this scoping if you want to access symbols in other scopes. For more details, see the section "Accessing symbols outside the current scope" on page 137.

Constants and number formats

Constants can be either real (floating-point) or integer constants. Negative constants start with a minus sign (-). If the number contains a decimal point or an *e* that introduces an exponent, it is a real number. For example,

```
123.4      456e34      123.45e-5
```

Integer-type constants are normally decimal, unless they start with a dollar sign ($) to indicate hexadecimal. Decimal integer constants must be between –2,137,483,648 and 2,147,483,647.

Hexadecimal constants must be between $00000000 and $FFFFFFFF.

Strings

A string is simply a group of characters surrounded by single quotes. For example,

```
'abc'
```

You can embed control characters in a string by preceding the decimal control character value with a #. For example,

```
'def'#7'xyz'
```

Operators and operator precedence

TDW supports all the Pascal expression operators.

The unary operators are of the highest precedence and are of equal priority.

@	Takes address of an identifier
^	Contents of pointer
not	Bitwise complement
typeid	Typecast
+	Unary plus, positive
–	Unary minus, negative

The binary operators are of a lower precedence than the unary operators. They are listed here in descending order (operators on the same line have the same priority):

* / **div** **mod** **and** **shl** **shr**

in + – **or** **xor**

< <= > >= = <>

The assignment operator (**:=**) has the lowest precedence; it returns a value.

Calling functions and procedures

You can refer to Pascal functions and procedures in expressions. For example, assume you have declared a function called *HalfFunc* that divides an integer by 2:

```
function HalfFunc(i:Integer): Real;
```

You can then choose Data | Evaluate/Modify and call *HalfFunc* as follows:

```
HalfFunc(3)
HalfFunc(10) = HalfFunc(10 div 2)
```

You can also call procedures, although not in an expression, of course. When you enter a procedure or function name by itself, TDW reports its address and declaration. To call a function or procedure that has no parameter, place a set of empty parentheses after the symbol name. For example,

`MyProc()`	Calls *MyProc*
`MyProc`	Reports *MyProc*'s address, and so on
`MyFunc = 5`	Compares address of *MyFunc* to 5
`MyFunc() = 5`	Calls *MyFunc* and compares returned value to 5

Assembler expressions

TDW supports the complete assembler expression syntax. An assembler expression consists of a mixture of symbols, operators, strings, variables, and constants. Each of these components is described in this section.

Assembler symbols

Symbols are user-defined names for data items and routines in your program. An assembler symbol name starts with a letter (*a-z, A-Z*) or one of these symbols: @ ? _ $. Subsequent characters in the symbol can contain the digits *0* to *9*, as well as these characters. The period (.) can also be used as the first character of a symbol name, but not within the name.

The special symbol *$* refers to your current program location as indicated by the CS:IP register pair.

Assembler constants

Constants can be either floating point or integer. A floating-point constant contains a decimal point and may use decimal or scientific notation. For example,

```
1.234      4.5e+11
```

Integer constants are hexadecimal unless you use one of the assembler conventions for overriding the radix:

Format	Radix
digitsH	Hexadecimal
digitsO	Octal
digitsQ	Octal
digitsD	Decimal
digitsB	Binary

If you want to end a hex number with a D or B, you must append an H to avoid ambiguity.

You must always start a hexadecimal number with one of the digits *0* to *9*. If you want to enter a number that starts with one of the letters *A* to *F*, you must first precede it with a *0* (zero).

Assembler operators

TDW supports most of the assembler operators. The first line in the list that follows shows the operators with the lowest priority, and the last line those operators with the highest priority. Within a line, all the operators have the same priority.

xxx PTR (BYTE PTR...)
. (structure member selector)
: (segment override)
OR XOR
AND
NOT
EQ NE LT LE GT GE
+ –
* / **MOD SHR SHL**
Unary + Unary –
OFFSET SEG
( ) [ ]

Variables can be changed using the = assignment operator. For example,

```
a = [BYTE PTR DS:4]
```

Format control

When you supply an expression to be displayed, TDW displays it in a format based on the type of data it is. TDW ignores a format control that is wrong for a particular data type.

If you want to change the default display format for an expression, place a comma at the end of the expression and supply an optional repeat count followed by an optional format letter. You can only supply a repeat count for pointers or arrays.

Character	Format
c	Displays a character or string expression as raw characters. Normally, nonprinting character values are displayed as some type of escape or numeric format. This option forces the characters to be displayed using the full IBM display character set.
d	Displays an integer as a decimal number.
f[#]	Displays as floating-point format with the specified number of digits. If you don't supply a number of digits, as many as necessary are used.
m	Displays a memory-referencing expression as hex bytes.
md	Displays a memory-referencing expression as decimal bytes.
p	Displays a raw pointer value, showing segment as a register name if applicable. Also shows the object pointed to. This is the default if no format control is specified.
s	Displays an array or a pointer to an array of characters as a quoted character string.
x or h	Displays an integer as a hexadecimal number.

C H A P T E R

10

Object-oriented debugging

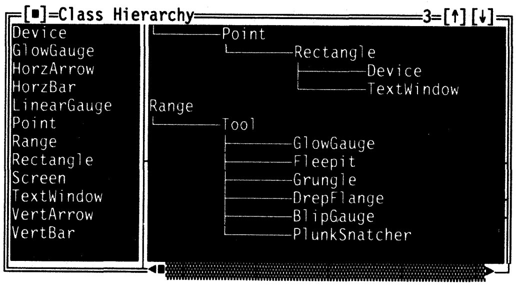
OOP

To meet the needs of the object-oriented programming revolution, TDW supports object-oriented Pascal. Besides extensions that let you trace into object methods and examine objects in the Evaluate/Modify dialog box and the Watches window, TDW comes equipped with a special set of windows and local menus specifically designed for objects and object types.

The Hierarchy window

TDW provides a special window for examining object type hierarchies. You can bring up the Hierarchy window by choosing View | Hierarchy.

Figure 10.1
The Hierarchy window



```
[■]=Class Hierarchy=========================3=[↑][↓]
Device        └──────Point
GlowGauge            └──────Rectangle
HorzArrow                   ├──────Device
HorzBar                     └──────TextWindow
LinearGauge   Range
Point         └──────Tool
Range                ├──────GlowGauge
Rectangle            ├──────Fleepit
Screen               ├──────Grungle
TextWindow           ├──────DrepFlange
VertArrow            ├──────BlipGauge
VertBar              └──────PlunkSnatcher
```

Use Tab to move between the two panes.

The Hierarchy window displays information on object *types* rather than instances. The left pane, the Object Type List pane, lists in alphabetical order the types used by the module being debugged. The right pane, the Hierarchy Tree pane, shows all object types in their hierarchies by using a line graphic that places the base type at the left margin of the pane and displays descendants beneath and to the right of the base type, with lines indicating descendant relationships.

The Object Type List pane

The left pane of the Hierarchy window provides an alphabetical list of all object types used by the current module. It supports an incremental matching feature to eliminate the need to scroll through large lists of types: When the highlight bar is in the left pane, simply start typing the name of the object type you're looking for. At each key press, TDW highlights the first type matching all keys pressed up to that point.

Press *Enter* to open an object type Inspector window for the highlighted type. Object type Inspector windows are described on page 151.

The Object Type List pane local menu

Press *Alt-F10* to display the local menu for the pane. You can use the control-key shortcuts if you've enabled hot keys with TDWINST (described in the online text file UTILS.TDW). This local menu contains two items: Inspect and Tree.

```
Inspect
Tree
```

Inspect

Displays an object type Inspector window for the highlighted type.

Tree

Moves to the right pane in which the hierarchy tree is displayed and places the highlight bar on the type that was highlighted in the left pane.

The Hierarchy Tree pane

The right pane displays the hierarchy tree for all object types used by the current module. Ancestor and descendant relationships are indicated by lines, with descendants to the right of and below their ancestors.

To locate a single object type in a complex hierarchy tree, go back to the left pane and use the incremental search feature; then choose the Tree command from the local menu to move back into the hierarchy tree. The matched type appears under the highlight bar.

When you press *Enter*, an object type Inspector window appears for the highlighted type.

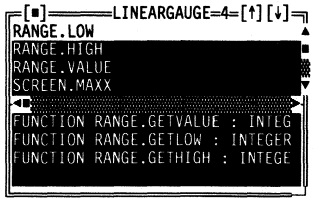
The Hierarchy Tree pane local menu

```
Inspect
```

The Hierarchy Tree pane local menu (*Alt-F10* in that pane) has only one item: Inspect. When you choose it, an object type Inspector window appears for the highlighted type. However, a faster and easier method is simply to press *Enter* when you want to inspect the highlighted type.

Object type Inspector windows

TDW provides a special type of Inspector window to let you inspect the details of an object type: the object type Inspector window. The window summarizes type information, but does not reference any particular instance. You display this window by bringing up the Object Hierarchy window (choose View | Hierarchy), selecting an object type, and pressing *Ctrl-I*.



```
[■]═══════LINEARGAUGE=4=[↑][↓]
RANGE.LOW
RANGE.HIGH
RANGE.VALUE
SCREEN.MAXX

FUNCTION RANGE.GETVALUE : INTEG
FUNCTION RANGE.GETLOW : INTEGER
FUNCTION RANGE.GETHIGH : INTEGE
```

Figure 10.2
An object type Inspector window

The window is divided horizontally into two panes, with the top pane listing the data fields of the type and the bottom pane listing the method names and (if the selected item is a function rather than a procedure) the function return type. Use *Tab* to move between the two panes of the object type Inspector window.

If the highlighted data field is an object type or a pointer to an object type, pressing *Enter* opens another object type Inspector window for the highlighted type. (This action is identical to choosing Inspect in the local menu for this pane.) In this way, complex nested structures of objects can be inspected quickly with a minimum of keystrokes.

For brevity's sake, method parameters are not shown in the object type Inspector window. To examine parameters, highlight the method and press *Enter*. A method Inspector window appears. The top pane of the window displays the code address for the object's implementation of the selected method, and the names and types of all its parameters. The bottom pane of the window indicates whether the method is a procedure or a function.

Pressing *Enter* from anywhere within the method Inspector window brings the Module window or the CPU window to the foreground, with the cursor at the code that implements the method being inspected.

As with standard inspectors, *Esc* closes the current Inspector window and *Alt-F3* closes them all.

The object type Inspector window local menus

Pressing *Alt-F10* brings up the local menu for either pane. If control-key shortcuts are enabled (through TDWINST), you can get to a local menu item by pressing *Ctrl* and the first letter of the item.

```
Inspect
Hierarchy
Show inherited   Yes
```

The Object Data Field (top) pane

The Object Data Field pane local menu contains these items:

Inspect

If the highlighted field is an object type or a pointer to one, a new object type Inspector window is opened for the highlighted field.

Hierarchy

Opens a Hierarchy window for the object type being inspected. The Hierarchy window is described on page 149.

Show Inherited

Yes is the default value of this toggle. When Show Inherited is set to *Yes*, TDW shows all data fields, whether they are defined within the type of the inspected object or inherited from an ancestor type. When it is set to *No*, TDW displays only those fields defined within the type being inspected.

The Object Method (bottom) pane

The local menu commands for the bottom Object Method pane are Inspect, Hierarchy, and Show Inherited.

Inspect

A method Inspector window is opened for the highlighted item. If you press *Ctrl-I* when the cursor is positioned over the address shown in the method Inspector window, the Module window is brought to the foreground with the cursor at the code that implements what is being inspected.

Hierarchy

Opens a Hierarchy window for the object type being inspected. The Hierarchy window is described on page 149.

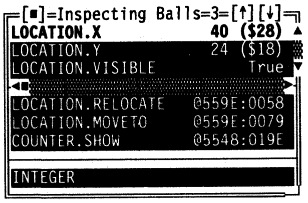
Show Inherited

Yes is the default value of this toggle. When it is set to *Yes*, all methods are shown, whether they are defined within the type being inspected or inherited from an ancestor. When it is set to *No*, only those methods are displayed that are defined within the object type being inspected.

Object instance Inspector windows

Object type Inspector windows provide information about object types, but say nothing about the data contained in a particular object instance at a particular time during program execution. TDW provides an extended form of the familiar record Inspector window specifically to inspect object instances.

Bring up this window by placing your cursor on an object instance in the Module window, then pressing *Ctrl-I.*



```
=[■]=Inspecting Balls=3=[↑][↓]=
LOCATION.X                40 ($28)
LOCATION.Y                24 ($18)
LOCATION.VISIBLE              True

LOCATION.RELOCATE     @559E:0058
LOCATION.MOVETO       @559E:0079
COUNTER.SHOW          @5548:019E

INTEGER
```

Figure 10.3
An object instance Inspector window

See the Turbo Pascal for Windows manuals for a description of the Virtual Method Table (VMT).

Most TDW data record Inspector windows have two panes: a top pane summarizing the record's field names and their current values, and a bottom pane displaying the type of the field highlighted in the top pane. An object instance Inspector window provides both of those panes, and also a third pane between them. This third pane summarizes the instance's methods, with the code address of each. (The code address takes into account polymorphic objects and the Virtual Method Table.)

The object instance Inspector window local menus

Each of the top two panes of the object instance Inspector window has its own local menu, displayed by pressing *Alt-F10* in that pane. Use the control-key shortcuts to get to individual menu items if you've enabled hot keys with TDWINST.

```
Range...
Change...
Methods          Yes
Show inherited   Yes

Inspect
Descend
New expression...
Type cast
Hierarchy
```

As with record Inspector windows, the bottom pane serves only to display the type of the highlighted field and doesn't have a local menu.

The local menu commands for the top pane, which summarizes the data fields for the selected item, are described here.

Range This command displays the range of array items. If the inspected item is not an array or a pointer, the item cannot be accessed.

Change By choosing this command, you can load a new value into the highlighted data field.

Methods This command is a *Yes/No* toggle, with *Yes* as the default condition. When it's set to *Yes*, methods are summarized in the middle pane. When it's set to *No*, the middle pane doesn't appear. This setting is carried over to the next Inspector window to be opened.

Show Inherited This command is also a *Yes/No* toggle. When it's set to *Yes*, all fields and all methods are shown, whether they are defined within the type being inspected or inherited from an ancestor type. When it's set to *No*, only those fields and methods defined within the type being inspected are displayed.

Inspect Choosing this command opens an Inspector window on the highlighted field. Pressing *Enter* over a highlighted field does the same thing.

Descend The highlighted item takes the place of the item in the current Inspector window. No new Inspector window is opened. However, you can't return to the previously inspected field, as you could if you had used the Inspect option.

☞ Use Descend to inspect a complex data structure when you don't want to open a separate Inspector window for each item.

New Expression This command prompts you for a new field or expression to inspect. The new item replaces the current one in the window; it doesn't open another window.

Type Cast Lets you specify a different data type for the item being inspected. This command is useful if the Inspector window contains a symbol for which there is no type information, as well as for explicitly setting the type for pointers.

Hierarchy When you choose this command, a Hierarchy window opens. For a full description of this window, see page 149.

The middle and bottom panes

The middle pane summarizes the methods of an object. The only difference between the Object Method pane's local menu and the local menu for the top pane is the absence of the Change command. Unlike data fields, methods cannot be changed during execution, so there is no need for this command. The bottom pane displays the type of the item highlighted in the upper two windows.

C H A P T E R

11

Using Windows debugging features

This chapter covers the features of TDW that give you access to Windows information and let you do the following:

- Log messages received and sent by your application's windows
- List the global heap
- List the local heap
- View the complete list of modules (including dynamic link libraries) loaded by Windows
- Debug dynamic link libraries (DLLs)
- See the contents of any protected-mode selector (in the CPU window)

Windows features

The features that support debugging of Windows programs are

- A view window, the Windows Messages window, which shows messages passed to windows in your program
- Three types of data you can display in the Log window:
  - The data segments in your program's local heap
  - The data segments in the global heap
  - A complete list of modules making up your program, including any dynamic link libraries (DLLs)

- Expression typecasting from memory handles to far pointers
- Support for debugging of DLLs in the Load Module Source or DLL Symbols window (choose View | Modules)
- The Selector pane of the CPU window, which allows you to see the contents of any protected-mode selector.

Logging window messages

To track messages being passed to your program's windows, choose the View | Windows Messages command to open the Windows Messages window. This window shows you the messages that Windows is passing to one or more windows in your program.

The Windows Messages window is composed of three panes, the Window Selection pane (top left), the Message Class pane (top right), and the Messages pane (bottom). The messages show up in the Messages pane.

The appearance of this window and the way you add application windows to it differ depending on whether you're working with an ObjectWindows application or a standard Windows application.

Selecting a window for a standard Windows application

If you're debugging a standard Windows application and you select View | Windows Messages, you see the following window:

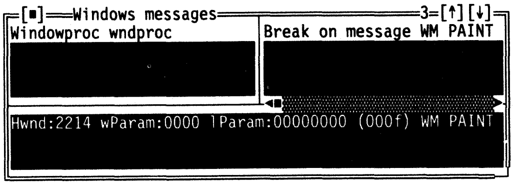

```
[■]═Windows messages═══════════════════3═[↑][↓]
Windowproc wndproc          Break on message WM PAINT
Hwnd:2214 wParam:0000 lParam:00000000 (000f) WM PAINT
```

Figure 11.1
The Windows Messages window for a standard Windows application

```
Add...
Remove
Delete all
```

Before you can log messages, you must first indicate which window you're logging messages for. You do this in the top left pane, the Window Selection pane. This pane's local menu (activated by pressing *Alt-F10*) lets you add a window selection, delete a window selection, or delete all window selections.

Adding a window selection for a standard Windows application

To add a window selection, you can either choose Add from the Window Selection pane local menu or begin typing in the pane. Either method brings up the Add Window dialog box.

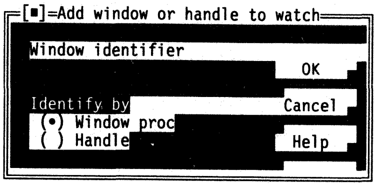


Figure 11.2
The Add Window dialog box for a standard Windows application

Adding the first window proc to this box also sets the message class to "Log all messages."

You can enter either the name of the object that processes messages for the window (select the Window Proc button) or a handle value (select the Handle button). Enter as many routine names or handle values as necessary to track messages for your windows.

It's easier to indicate the window by the name of the routine that processes its messages (for example, *WndProc*) because you can enter a routine name any time after loading your program.

If you prefer to use a handle variable name, you must first step through the program past the line where the handle variable is assigned a handle. (Use the *F7* or *F8* key to single-step through the program.) If you try to enter the variable name before stepping past its assignment statement, TDW will not let you.

Selecting a window for an ObjectWindows application

If you're debugging an ObjectWindows application and you select View | Windows Messages, by default you see the standard Windows Messages dialog box in Figure 11.1. This dialog box works the same for ObjectWindows programs as for standard Windows programs, except that you can't use a Windows procedure name. Instead, you must use the handle to the window object for the window whose messages you want to log or break on.

Obtaining a window handle

Before you can use the handle of a window object, you must run your program past the point where the handle is initialized. You can use a number of techniques to do this.

- It's simplest just to run your application and exit back to TDW with *Ctrl-Alt-SysRq*.
- Another possibility is to set a breakpoint in a message-handling routine in your program (such as a routine that handles WM_MOUSEMOVE messages), run the program, and then perform the action in the window that triggers the breakpoint (for example, moving the mouse).
- If you're having major problems with the window itself (such as an unrecoverable application error (UAE) that comes up when the window is first displayed), you'll have to go to greater lengths to obtain the window handle.

  Because the handle is initialized by the ObjectWindows method *SetupWindow*, you can only get the window handle after this method has executed. The easiest way to to this is to set a breakpoint after the call to *TWindow.SetupWindow*.

  However, not all programs contain this call explicitly. For example, the program TDODEMO does not contain a direct *SetupWindow* call. In order to obtain the window handle, you must override the method for the TDODEMO window object *ScribbleWindow* as follows:

  ```
  type
     ScribbleWindow=object(TWindow)
        procedure SetupWindow;  virtual;
  end;
  ```

  Once this declaration has been made, you can create a dummy method with a call to *SetupWindow*:

  ```
  procedure ScribbleWindow.SetupWindow;
  begin
     TWindow.SetupWindow;
  end;
  ```

  Next, position the cursor on the end statement after the call to statement and press *F4* to run the program to the point where the handle of the window, dialog box, or control is initialized. In this example, you'd position the cursor on the **end** of the function *SetupWindow*.

  If you get the UAE before *SetupWindow* is called, then the problem lies before the creation of the window.

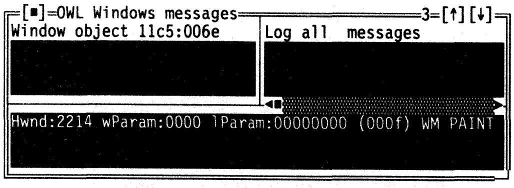
Once the handle is initialized and you've returned to TDW, you can obtain its value by choosing Data | Inspect and entering the name of the associated window object (in TDODEMO, *MyApp^.MainWindow*). Look for the data member *HWindow* and copy it into the Clipboard (press *Shift-F3*). You can then paste the *contents* of *HWindow* as a handle into the Add dialog box of the Window Messages window's top left pane (press *Shift-F4* in the dialog box's text entry box).

Specifying a window with ObjectWindows support enabled

See the file UTILS.TDW for information on setting options in TDWINST.

If you run the TDW configuration program TDWINST, you can turn on support in TDW for ObjectWindows window messages. With this option on, you can use the names of windows objects as they're declared in your application. Choosing View | Windows Messages with the OWL option on displays the following screen:

Figure 11.3
The Windows Messages window with ObjectWindows support enabled



```
[■]=OWL Windows messages=3=[↑][↓]
Window object 11c5:006e    Log all  messages
Hwnd:2214 wParam:0000 lParam:00000000 (000f) WM PAINT
```

```
Add...
Remove
Delete all
```

Before you can log messages, you must first indicate which window, dialog box, or dialog control you're logging messages for. You do this in the top left pane, the Window Selection pane. This pane's local menu (activated by pressing *Alt-F10*) lets you add a window object, delete a window object, or delete all window objects.

Adding a window with ObjectWindows support enabled

Before adding a window object, you must run your program past the point where the window object is initialized. Typically, the object is initialized in a statement like the one in the following procedure definition from TDODEMO:

```
procedure CScribbleApplication.InitMainWindow;
begin
  MainWindow := New(PCScribbleWindow,Init(nil, 'Scribble With
Color!'));
end;
```

Position the cursor on the line after the initialization statement and press *F4* to run the program to the point where the window, dialog box, or control is initialized. In this example, you'd position the cursor on the **end** keyword.

Once the window object is initialized, you can add it to the Window Selection pane. To add the object, either choose Add from the Window Selection pane local menu or begin typing the object's name in the pane. Either method brings up the Add Window dialog box.

If you're not in the routine where the object is declared, you have to override scope to access it. For example, in TDODEMO, *MainWindow* is a field of *CSApp* (because *CSApp* is of type *CScribbleApplication*, which is derived from *TApplication*, which has a field called *MainWindow*). Since *CSApp* is declared in the main program, it's available globally, so the scope override statement is `CSApp.MainWindow`.

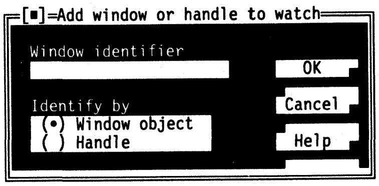


Figure 11.4
The Add Window dialog box with ObjectWindows support enabled

```
[■]=Add window or handle to watch
Window identifier
Identify by
(•) Window object
( ) Handle
```

Adding the first object to this pane also sets the message class to "Log all messages."

You can enter either the name of the object that processes messages for the window, dialog box, or control (select the Window Object button) or a handle value (select the Handle button). Enter as many object names or handle values as necessary to track messages for your windows.

Deleting a window selection

Deleting a window selection from the Window Selection pane works the same for both types of applications. To delete, move the cursor to the item, then either bring up the local menu and choose Remove or press the *Delete, Ctrl-Y,* or *Ctrl-R* key.

To delete all selections, choose Delete All from the local menu.

Specifying a message class and action

```
Add...
Remove
Delete all
```

The top right pane is the Message Class pane. Its local menu, identical to that of the Window Selection pane, allows you to add a message class, remove a message class, or delete all classes you have added.

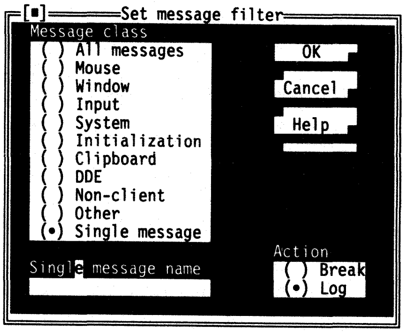
You must specify a window procedure or handle in the Window Selection pane before you can add a message class in this pane.

If you don't indicate a specific message or class of messages to watch, TDW watches all messages sent to the window procedure or handle.

Adding a message class

To add a message class, choose Add from the Message Class pane local menu. TDW displays the following dialog box:

Figure 11.5
The Set Message Filter dialog box



```
[■]══════Set message filter══════
Message class
( ) All messages                OK
( ) Mouse                       Cancel
( ) Window                      Help
( ) Input
( ) System
( ) Initialization
( ) Clipboard
( ) DDE
( ) Non-client
( ) Other
(•) Single message
                                Action
Single message name             ( ) Break
                                (•) Log
```

The Set Message Filter dialog box prompts you both for a message class to track and an action to be performed when a message in that class is received.

TDW by default logs all messages starting with *WM_*. Because so many messages come in, you'll probably want to narrow the focus by selecting one of the classes in the Message Class list. You can add only one class at a time, so if you need to track messages from multiple classes, you have to use the Add option for each class you want to set.

The following table describes the message classes:

Table 11.1
Windows message classes

Message class	**Description**
All Messages	All window messages
Mouse	Messages generated by a mouse event (for example, WM_LBUTTONDOWN and WM_MOUSEMOVE)
Window	Messages from the window manager (for example, WM_PAINT and WM_CREATE)
Input	Messages generated by a keyboard event or by the user's accessing a System menu, scroll bar, or size box (for example, WM_KEYDOWN)
System	Messages generated by a system-wide change (for example, WM_FONTCHANGE and WM_SPOOLERSTATUS)
Initialization	Messages generated when an application creates a dialog box or a window (for example, WM_INITDIALOG and WM_INITMENU)

Table 11.1: Windows message classes (continued)

Clipboard	Messages generated when one application tries to access the Clipboard of a window in another application (for example, WM_DRAWCLIPBOARD and WM_SIZECLIPBOARD)
DDE	Dynamic Data Exchange messages, generated by applications' communicating with one another's windows (for example, WM_DDE_INITIATE and WM_DDE_ACK)
Non-client	Messages generated by Windows to maintain the non-client area of an application window (for example, WM_NCHITTEST and WM_NCCREATE)
Other	Any messages that don't fall into any of the other categories, such as owner draw control messages and multiple document interface messages
Single Message	Any single message you want to log or break on

To track a single message, choose Single Message and enter the message name or the message number as a decimal number. If you enter a message name, be sure to use all capital letters.

The default action is to put the messages in the log. The other action you can perform, having the program break when it receives a message, is equivalent to setting a breakpoint for a message.

Setting a message breakpoint

For example, if you want to track the WM_PAINT message and have the program stop every time this message is sent to a window you chose in the Window Selection pane, do the following:

1. Select the top right pane, the Message Class pane.
2. Bring up the local menu, then choose Add.
3. From the dialog box, select Break from the Action radio buttons and Single Message from the Message Class radio buttons.
4. Enter *WM_PAINT* in the Message Name input box, then press *Enter.*

Figure 11.1 on page 158 shows how the Windows Messages window looks after you have made these selections and a message has come in.

Deleting a message class

To delete a message class, move the cursor to the item, then either bring up the local menu and choose Remove or press one of the following keys: *Delete*, *Ctrl-R*, or *Ctrl-Y*.

To delete all classes, choose Delete All from the local menu or press *Ctrl-D*.

The default class, Log all messages, appears after you have deleted all classes. You cannot delete this class using Remove or Delete All command.

Window message tips

If you're displaying messages for more than one window, do not log all messages. Instead, log specific messages or a specific message class for each window. If you log all messages, the large number of messages being transferred between Windows and TDW might cause your system to hang.

When setting a break on Mouse class messages, be aware that a *mouse down* message must be followed by a *mouse up* message before the keyboard becomes active again. This restriction means that when you return to the application, you might have to press the mouse button several times to get Windows to receive a *mouse up* message. You'll know that Windows has received the message when you see it in the lower pane of the Windows Messages window.

If you enter a handle name but indicate that it's a procedure, TDW accepts your input and doesn't complain. However, when you run your program, TDW does not log any messages. If TDW isn't logging messages after you've set a handle, reenter the handle and be sure to select the Handle button.

Viewing messages

```
Send to log window  No
Erase log
```

Window messages show up in the lower pane of the Windows Messages window. This pane can hold up to 200 messages.

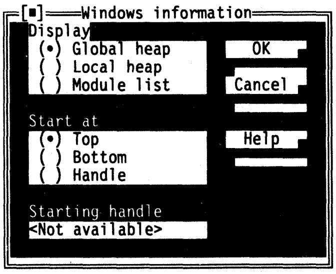
If you want to save the messages to a file, you have to open a log file for the Log window (use View | Log File, then choose Open Log File from the local menu). Then switch back to the Messages pane and change the Send To Log Window entry on the local menu to *Yes*.

If you want to clear the pane of all messages, choose Erase Log from the local menu. Any messages written to the Log window will not be affected by this command.

Obtaining memory and module lists

To list the contents of the global or local heap or the modules for your Windows program, first bring up the Log window with View | Log, then access the local menu. The last command on the Log window local menu is Display Windows Info. Choosing that command displays the Windows Information dialog box, from which you can pick the type of list you want to display and where to start the list.

Figure 11.6 The Windows Information dialog box


```
[■]═══Windows information═══
Display
 (•) Global heap        OK
 ( ) Local heap
 ( ) Module list        Cancel

Start at
 (•) Top                Help
 ( ) Bottom
 ( ) Handle

Starting handle
<Not available>
```

If you select the Global Heap option, you can choose to display the list from top to bottom, from bottom to top, or from a location indicated by a starting handle.

A starting handle is the name of a global memory handle set in your application by a call to a Windows memory allocation routine like *GlobalAlloc*. Picking a starting handle causes TDW to display the object at that handle as well as the next four objects that follow it in the heap.

Listing the contents of the global heap

The *global heap* is the global memory area Windows makes available to all applications. If you allocate resources like icons, bit maps, dialog boxes, and fonts, or if you allocate memory using the *GlobalAlloc* function, your application is using the global heap.

To see a list of the data objects in the global heap, select the Global Heap radio button in the Windows Information dialog box, then choose OK. The data objects will be listed in the Log window.

Because this list is likely to exceed the number of lines the Log window can write (the default is 50 lines), you should either write the contents to a log file (use the Log window local menu) or increase the number of lines the Log window can use (use TDWINST). The maximum number of lines you can set is 200.

The following table shows two lines of sample global heap output followed by an explanation of each field in the sample output:

Table 11.2
Format of a global heap list

Sample global heap output

```
0EC5        00000040b PDB (0F1D)
053E (053D) 00002DC0b GDI      DATA MOVEABLE LOCKED=00001 PGLOCKED=0001
```

Field	Description
0EC5 053E	Either a handle to the memory object, expressed as a 4-digit hex value, or the word *FREE,* indicating a free memory block.
(053D)	A memory selector pointing to an entry in the global descriptor table. The selector isn't displayed if it's the same value as the memory handle.
00000040b 00002DC0b	A hexadecimal number representing the length of the segment in bytes.
PDB GDI	The allocator of the segment, usually an application or library module. A PDB is a process descriptor block, also known as a program segment prefix (PSP).
(0F1D)	A handle indicating the owner of a PDB.
DATA	The type of memory object. The types are DATA — Data segment of an application or DLL CODE — Code segment of an application or DLL PRIV — Either a system object or global data for an application or DLL
MOVEABLE	A memory allocation attribute. An object can be FIXED, MOVEABLE, or MOVEABLE DISCARDABLE.
LOCKED=00001	For a moveable or moveable-discardable object, the number of locks on the object that have been set using either the *GlobalLock* or *LockData* function.
PGLOCKED=0001	For 386 Enhanced mode, the number of page locks on the object that have been set using the *GlobalPageLock* function. With a page lock set on a memory object, Windows can't swap to disk any of the object's 4-kilobyte pages.

Listing the contents of the local heap

The local heap is a private memory area for the application. It is not accessible to other Windows applications, including other instances of the same application.

A program doesn't necessarily have a local heap. Windows creates a local heap if the application uses the *LocalAlloc* function.

To see a list of the data objects in the local heap, select the Local Heap radio button in the Windows Information dialog box, then choose OK. The data objects will be listed in the Log window.

☞ The list can easily exceed the default length of the log window. See the caution in the previous global heap section (page 166) about using a log file or increasing the number of lines that can be written in the Log window.

The following table shows a typical line of local heap output followed by an explanation of its format:

Table 11.3
Format of a local heap list

Local heap output

```
05CD:  0024  BUSY  (10AF)
```

Field	**Description**	
05CD:	The object's offset in the local data segment	
0024	The length of the object in bytes	
BUSY	The disposition of the memory object, as follows:	
	FREE	An unallocated block of memory
	BUSY	An allocated object
(10AF)	A local memory handle for the object	

Obtaining a list of modules

To see a list of the task and DLL modules that have been loaded by Windows, select the Module List radio button in the Windows Information dialog box, then choose OK. The modules will be listed in the Log window.

☞ The list can easily exceed the default length of the log window. See the caution in the global heap section (page 166) about using a log file or increasing the number of lines that can be written in the Log window.

The following table shows three sample lines of a module list followed by an explanation of the last line in the list:

Table 11.4
Format of a Windows module list

Sample module list output

```
0985 TASK TDW      C:\TPW\TDW.EXE
0E2D DLL  TDWIN    C:\WINDOWS\TDWIN.DLL
0EFD TASK GENERIC  C:\TPW\GENERIC.EXE
```

Field	Description
0EFD	A handle for the memory segment, expressed as a 4-digit hex value.
TASK	The module type. A module can be either a task or a DLL.
GENERIC	The module name.
C:\TPW\GENERIC.EXE	The path to the module's executable file.

Debugging dynamic link libraries (DLLs)

A DLL is a library of routines and resources that is linked to your Windows application at run time instead of at compile time. This run-time linking allows multiple applications to share a single copy of routines, data, or device drivers, thus saving on memory use. When an application that uses a DLL starts up, if the DLL is not already loaded into memory, Windows loads it in so the application can access the DLL's entry points.

TDW can load a DLL that doesn't have a symbol table, but only into a CPU window.

When you load an application into TDW that has DLLs linked into it, TDW determines which of these DLLs, if any, have symbol tables (were compiled with the debugging option turned on) and tracks these DLLs for you. If, during execution of your application, TDW encounters a call to an entry point for one of these DLLs, TDW loads the symbol table and source for that DLL and positions the module line marker at the beginning of the DLL routine called by your application. The DLL is then available in the Module window just as your application source code was.

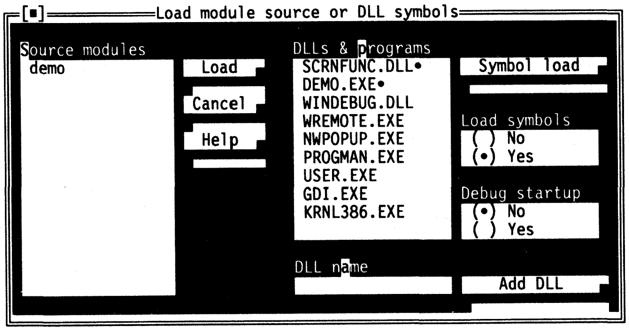
When the DLL routine exits, if possible, TDW reloads your application's source code and positions the line marker on the next statement after the call to the DLL entry point.

☞ If you are tracing through the program using *F7* or *F8*, it might not be possible for TDW to return you to the calling routine in your program because the DLL might return through a Windows function call. In this case, your program just runs as though you had pressed *F9*. This behavior is common in DLL startup code. To force a return to your application, before tracing in your application to the DLL call, set a breakpoint in your application on the line after the call to the DLL. When debugging DLL startup code, set the breakpoint on the first line of your application.

Because so much of DLL debugging is automatic with TDW, you never have to specify which DLLs to load. However, you might want to perform other tasks, such as:

- Adding a DLL to the list of DLLs
- Setting breakpoints, watches, and so on, in a DLL
- Specifying which DLLs TDW is *not* supposed to load symbols for
- Debugging DLL startup code

To perform any of these tasks, you have to access the Load Modules or DLLs dialog box by using the View | Modules command. (Pressing *F3* will also bring up this dialog box.)



```
[■]          Load module source or DLL symbols

Source modules            DLLs & programs
demo            Load      SCRNFUNC.DLL•    Symbol load
                Cancel    DEMO.EXE•
                Help      WINDEBUG.DLL
                          WREMOTE.EXE      Load symbols
                          NWPOPUP.EXE      ( ) No
                          PROGMAN.EXE      (•) Yes
                          USER.EXE
                          GDI.EXE          Debug startup
                          KRNL386.EXE      (•) No
                                           ( ) Yes
                          DLL name
                                           Add DLL
```

Figure 11.7
The Load Modules or DLLs dialog box

Using the Load Modules or DLLs dialog box

This dialog box enables you to do two things:

- Change to another source module of your application
- Perform operations on DLLs and .EXE files (such as loading in a symbol file and source file)

Changing source modules

If you're debugging an application consisting of multiple source modules linked into one .EXE file and you need access to a module of the application other than the one currently in the Module window, you can bring up the Load Modules or DLLs dialog box and pick one of the modules in the list on the left, the Source Modules list.

To pick a module, highlight it, and then either press *Enter*, click twice with the mouse, or choose the Load button. TDW displays the Module window with the new source module in it.

Working with DLLs and programs

When you're debugging an application that has one or more DLLs associated with it (as does any Windows application) and you bring up the Load Modules or DLLs dialog box, you see in the DLLs & Programs list (the list on the right) a list of DLLs and .EXE files (as well as all the .DRV and .FON files currently loaded into Windows). This list includes all DLL and .EXE files Windows currently has loaded, as well as all DLLs that get started when your application starts up. It does not include any DLLs your application starts by using a *LOADLIBRARY* call unless one of these DLLs is already loaded by your program or by Windows.

The items at the top of this list, marked on the right with an oval, are your application's .EXE file and the DLLs your application calls. If you make no changes, TDW automatically attempts to load in the symbol table and source for each marked DLL whenever your application makes a call to that DLL. In addition, TDW automatically attempts to load the symbol table and source of any DLL your application starts with a *LOADLIBRARY* call, even though the DLL might not appear on the list at first. (It will appear there after TDW loads it.)

The buttons to the right of this list perform operations on the DLL or application you have highlighted. The text entry box underneath the list lets you add a DLL to the list. You can use these features as follows:

Table 11.5
DLLs & Programs list dialog box controls

Button	**Description**
Symbol load	Load in the symbol table and source files for the DLL or application, regardless of the Load Symbols setting. This command overrides the Load Symbols setting and changes the contents of the Module Window so you can set breakpoints, window messages, and so on for the DLL or application.

Table 11.5: DLLs & Programs list dialog box controls (continued)

Load symbols (No/Yes)	Choose whether to load the DLL symbol table and source when the application makes a call to the DLL. You might use this option to prevent TDW from loading the symbol table and source of a DLL that you don't need to debug. The default setting is *Yes*. Choosing Yes puts an oval next to the DLL name. When you reload a program, Load Symbols is set to Yes for all DLLs and modules, even for DLLs or modules that were previously set to No.
Debug startup (No/Yes)	Choose whether to debug startup code for the DLL. The default setting is No. Choosing Yes puts double exclamation marks (!!) next to the module or DLL. These buttons are used for DLLs only. To debug application startup code, start TDW with the -l command-line option.
DLL Name	Enter the name of a DLL that isn't on the DLLs & Programs list so you can add it to the list. You can use any file extension you want. Adding a DLL to the list enables you to use one of the previous three commands on it. You can use a full path name if necessary.
Add DLL	Add the DLL in the text entry box to the DLLs & Programs list. Any DLL you add manually has both Load Symbols and Debug Startup set to Yes.

Adding a DLL to the DLLs & Programs list

Before you can set debug options, debug DLL startup code, or prevent TDW from loading a DLL's symbol table and source, the DLL must first be in the DLLs & Programs list. A DLL accessed by your application might not be in this list because, just after your application loads, TDW only knows about DLLs that are linked into the startup code of your application. Your application can also start a DLL explicitly by using the Windows *LOADLIBRARY* function; TDW won't know about it until your application calls *LOADLIBRARY*.

There are two different types of startup code mentioned in this section: your application's startup code and DLL startup code.

When your application starts a DLL, the DLL's startup code is then executed. Some DLLs are started before your application's startup code runs, and some are started during or afterwards. There are also two types of DLL startup code, explained later under "Debugging DLL startup code."

If you want to add a DLL to the DLLs & Programs list, bring up the Load Modules or DLLs dialog box (press *F3* or choose View | Modules), move to the DLL Name text entry box, enter the name of the DLL (enter the full path if necessary), then press the Add DLL button to add it to the list.

Setting debug options in a DLL

If you want to set breakpoints or watches or some other debug option for a DLL, bring up the Load Modules or DLLs dialog box (press *F3* or choose View | Modules), highlight the DLL on the DLLs & Programs list, then choose Symbol Load to bring up the DLL in a Module window. Once you're in the Module window, you can perform your operations on the DLL.

Controlling TDW's loading of DLL symbol tables

By default, TDW loads in the symbol table and source of every DLL that your application accesses, but only if the DLL has a TDW-compatible symbol table. A DLL has a symbol table compatible with TDW if it was compiled with debugging information turned on and the compiler was a Borland language compiler.

Because it takes time to load in a DLL's symbol table and then load in the original application's symbol table once the DLL routine has finished, you might want to disable TDW's default operation for DLLs you don't want to debug. To prevent TDW from loading a DLL's symbol table, bring up the Load Modules or DLLs dialog box (press *F3* or choose View | Modules), find the DLL on the DLLs & Programs list, highlight it, and then push the Load Symbols No button.

Debugging DLL startup code

By default, TDW does not debug DLL startup code and only loads a DLL's symbol table when your application makes a call to a DLL entry point. TDW then brings up the Module or CPU window with the current line marker at the beginning of the DLL routine called by the application.

TDW debugs DLL startup code if you tell it to. You can use TDW to debug either of two types of DLL startup code:

What kind of startup code are you debugging?

- The initialization code immediately following *LibMain* (the default)
- The assembly-language code linked into the DLL that does initial startup and contains emulated math packages for the size model the DLL is running in (selected by starting TDW with the **–l** command-line option)

After you specify startup debugging for one or more of the DLLs in your application, TDW loads in the symbol table for each DLL either when your application startup code starts the DLL or when your application makes a *LOADLIBRARY* call.

Is your application already loaded?

If you try to load your application and then set startup debugging, TDW might not behave as you expect, since some or all of the DLLs might already have been loaded. Therefore, you should set startup debugging by doing either of the following:

- Setting the DLLs before you load your application
- Loading your application, indicating the DLLs for startup debugging, and then restarting your application (*Ctrl-F2* or Run | Program Reset)

Doing startup code debugging

With all these preliminaries in mind, use the following steps to specify startup debugging for one or more DLLs and to debug those DLLs' startup code:

1. Bring up the Load Modules or DLLs dialog box (press *F3* or choose View | Modules).
2. If no program is loaded, skip to step 5. Otherwise, find a DLL on the DLLs & Programs list and highlight it.
3. Select the Debug Startup Yes button.
4. Repeat steps 2 and 3 until you've set startup debugging for all DLLs you're interested in.
5. If a DLL you want isn't on the list or there are no DLLs on the list (because you haven't loaded your application yet), use the DLL Name text entry box to enter each DLL name and add it to the list using the Add DLL button.
6. When you've set all the DLLs for which you want to debug startup code, choose either File | Load to load in your application (if you haven't loaded it yet) or Run | Program Reset (*Ctrl-F2*) to reload your application (if you loaded it before setting startup debugging).

7. Before you run the application, you should set breakpoints to guarantee that the DLLs will return to your application after the startup code executes. With your application's source code in the Module window,

   a. Set a breakpoint on the first line of your application.
   b. If you're debugging startup code for any DLLs loaded with *LOADLIBRARY* calls, set a breakpoint on the first line of code after each of these calls.

8. As your application starts each DLL, TDW puts you either in the Module window at the DLL's *LibMain* (the default) or in the CPU window at the start of the assembly code listing for the startup library (because you ran TDW using the **–l** option).
9. When you've finished debugging startup code for a DLL, press *F9* to run through the end of the startup code and return to the application. If you've specified any more DLLs for startup code debugging, TDW displays startup code for those DLLs when your application starts them.

Be sure to run to the end of a DLL's startup code before reloading the current application or loading a new one. If you don't, the partially executed DLL startup code might cause Windows to hang, forcing you to reboot.

Converting memory handles to addresses

Windows uses memory handles instead of addresses for memory objects because it performs its own memory management and can change the physical location in memory of an object. If you need the actual address referred to by a memory handle, you can use the TDW built-in typecast symbols **lh2fp** (for a local handle) and **gh2fp** (for a global handle) to dereference the handle.

You use these typecasting symbols in TDW just as you use the regular Turbo Pascal typecasting symbols for pointers. For example, you could cast the local memory handle *HLocalMemory* using two methods:

- You could use the Data | Inspect window to evaluate the expression `lh2fp(HLocalMemory)`.
- You could use the Type Cast command in the Inspector local window and enter *lh2fp* as the type.

In either case, the expression evaluates to the first character of the memory block pointed to by HLocalMemory.

You could also use either of these techniques to do a more complicated cast—for example, a two-stage cast from a handle into a character pointer into a pointer to the data in memory, as follows:

```
Mystruct(lh2fp(HLocalMemory))^
```

C H A P T E R

12

Assembler-level debugging

For more information on assembler-level debugging, see the file DOC\ASMDEBUG.TDW in the language compiler directory on your hard disk.

This chapter is for programmers who need to take a lower-level look at their code. It gives a brief introduction to the CPU window and the six panes of that window. You can also get online Help information about any pane of this window and its local menu by positioning the cursor in the pane and pressing *F1*.

When source debugging isn't enough

When you're debugging a program, most of the time you refer to data and code at the source level; you refer to symbol names exactly as you typed them in your source code, and you proceed through your program by executing pieces of source code.

Sometimes, however, you need information you can't get from the source code Module window, such as

- looking at the contents of an area in memory referenced by a protected-mode selector
- looking at the exact instructions that the compiler generated for a line of source code, as well as the contents of the stack and CPU registers
- tracing through Windows code to find where your program stopped

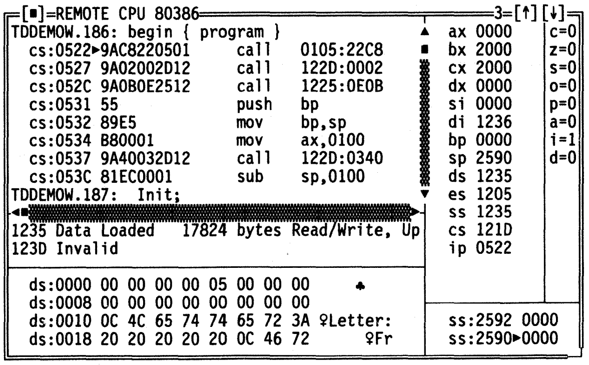
To perform any of these functions, you have to use the CPU window. In addition, it helps to be familiar with Windows' use of memory and to have knowledge of both the 80x86 family of processors and the machine instructions the compiler generates for your source code. Because many excellent books are available about the internal workings of the CPU, we won't go into that in detail here. You can quickly learn how the compiler turns your source code into machine instructions by looking at the instructions generated for each line of source code.

The CPU window

The CPU window shows you the entire state of the CPU. You can

- examine and change the bits and bytes that make up your program's code and data
- access any area of memory referenced by a selector
- use the built-in assembler in the Code pane to patch your program temporarily by entering instructions exactly as you would type assembler source statements
- access the underlying bytes of any data structure, display them in a number of formats, and change them

Figure 12.1
The CPU window



```
[■]=REMOTE CPU 80386                                3=[↑][↓]
TDDEMOW.186: begin { program }                  ▲  ax 0000   c=0
  cs:0522►9AC8220501     call   0105:22C8       ■  bx 2000   z=0
  cs:0527 9A02002D12     call   122D:0002          cx 2000   s=0
  cs:052C 9A0B0E2512     call   1225:0E0B          dx 0000   o=0
  cs:0531 55             push   bp                 si 0000   p=0
  cs:0532 89E5           mov    bp,sp              di 1236   a=0
  cs:0534 B80001         mov    ax,0100            bp 0000   i=1
  cs:0537 9A40032D12     call   122D:0340          sp 2590   d=0
  cs:053C 81EC0001       sub    sp,0100            ds 1235
TDDEMOW.187:  Init;                             ▼  es 1205
◄■                                              ►  ss 1235
1235 Data Loaded   17824 bytes Read/Write, Up      cs 121D
123D Invalid                                       ip 0522

  ds:0000 00 00 00 00 05 00 00 00      ♣
  ds:0008 00 00 00 00 00 00 00 00
  ds:0010 0C 4C 65 74 74 65 72 3A ♀Letter:         ss:2592 0000
  ds:0018 20 20 20 20 20 0C 46 72      ♀Fr         ss:2590►0000
```

Open a CPU window by choosing View | CPU from the menu bar. Depending on what you're viewing in the current window, the new CPU window comes up positioned at the appropriate code, data, or stack location, thus providing a convenient method for taking a low-level look at the code, data, or stack location your cursor is currently on.

The following table shows where your cursor is positioned when you choose the CPU command:

Current window	CPU pane	Position
Stack window	Stack	Top of stack frame for highlighted item
Module window	Code	Address of item
Variable window	Data/Code	Address of item
Watches window	Data/Code	Address of item
Inspector window	Data/Code	Address of item
Breakpoint (if not global)	Code	Breakpoint address
Other area	Code	Current CS:IP

TDW also puts you in the CPU window if TDW regains control from your application when the code being executed is Windows code or DLL code with no debugging information.

The CPU window has six panes. To go from one pane to the next, press *Tab* or *Shift-Tab*, or click the pane with your mouse. The line at the top of the CPU window shows what processor type you have (8086, 80286, 80386, or 80486).

- The top left pane (Code pane) shows the disassembled program code intermixed with the source lines.
- The second top pane (Register pane) shows the contents of the CPU registers.
- The top right pane (Flags pane) shows the state of the eight CPU flags.
- The middle left pane (Selector pane—below the Code pane) shows all Windows selectors and indicates the general contents of each.
- The bottom left pane (Data pane—below the Selector pane) shows a raw hex dump of any area of memory you choose.
- The bottom right pane (Stack pane) shows the contents of the stack.

As with all windows and panes, pressing *Alt-F10* pops up the pane's local menu. If control-key shortcuts are enabled, pressing the *Ctrl* key with the highlighted letter of the desired local menu command executes the command.

While in the Code pane, Data pane, and Stack pane of the CPU window, it's possible to scroll outside the current protected-mode segment. Since addresses lying outside the protected-mode segment are considered invalid by the operating system, these addresses will display data as question marks while inside the CPU window panes.

In the Code, Data, and Stack panes, you can press *Ctrl* ← and *Ctrl* → to shift the starting display address of the pane by 1 byte up or down. Pressing these keys is easier than using the Goto command if you just want to adjust the display slightly.

The Code pane

An arrow (▸) shows the current program location (CS:IP).

This pane shows the disassembled instructions at an address that you choose.

There are two ways of choosing an address:

- Use the local menu Goto, Origin, Follow, Caller, or Previous command.
- Position on a code selector in the Selector pane, then choose Examine to display the contents of the selector in the Code pane.

The left part of each disassembled line shows the address of the instruction. The address is displayed either as a hex segment and offset, or with the segment value replaced with the CS register name if the segment value is the same as the current CS register. If the window is wide enough (zoomed or resized), the bytes that make up the instruction are displayed. The disassembled instruction appears to the right.

If the highlighted instruction in the Code pane references a memory location, the memory address and its current contents are displayed on the top line of the CPU window. This feature lets you see both where an instruction operand points in memory and the value that is about to be read or written over.

The disassembler

The Code pane automatically disassembles and displays your program instructions. If an address corresponds to either a global symbol, static symbol, or a line number, the line before the disassembled instruction displays the symbol if the Mixed display mode is set to *Yes*. Also, if there is a line of source code that corresponds to the symbol address, it is displayed after the symbol.

Global symbols appear simply as the symbol name. Static symbols appear as the module name, followed by a period (.), followed by the static symbol name. Line numbers appear as the module name, followed by a period (.), followed by the decimal line number.

When an immediate operand is displayed, you can infer its size from the number of digits: A byte immediate has 2 digits, and a word immediate has 4 digits.

The Register and Flags panes

The Register pane, which is the top pane to the right of the Code pane, shows the contents of the CPU registers.

The top right pane is the Flags pane, which shows the state of the eight CPU flags. The following table lists the different flags and how they are shown in the Flags pane:

Letter in pane	Flag name
c	Carry
z	Zero
s	Sign
o	Overflow
p	Parity
a	Auxiliary carry
i	Interrupt enable
d	Direction

You can use the local menu of the Register pane to increment or decrement a register by 1, to set the register to 0, to change the register, or to toggle between displaying the register as 16-bit or 32-bit values (requires an 80386 processor or greater).

The local menu of the Flags pane allow you to toggle the flag between 0 and 1.

The Selector pane

This pane shows a list of protected-mode selectors and indicates some information about each one.

A selector can be either valid or invalid. If valid, the selector points to a location in the protected-mode descriptor table corresponding to a memory address. If invalid, the selector is unused.

For a valid selector, the pane shows the following:

- if the contents are data or code
- if the memory area the selector references is loaded (present in memory) or unloaded (swapped out to disk)
- the length of the referenced memory segment in bytes

If the selector references a data segment, there's additional information on the access rights (Read/Write or Read only) and the direction the segment expands in memory (Up or Down).

The Selector pane local menu

At the Selector pane, press *Alt-F10* to pop up the local menu or, if control-key shortcuts are enabled, use the *Ctrl* key with the highlighted letter of a command to access it.

You can use the local menu of the Selector pane to go to a new selector (the Selector command) or see the contents of the selector currently highlighted in the Selector pane (the Examine command). The contents display in either the Code pane or the Data pane, depending on their nature.

```
Selector
Examine...
```

Selector

Prompts you to type a selector to display in the pane. You can use full expression syntax to enter the selector. If you enter a numeric value, TDW assumes it is decimal unless you use the syntax of the current language to indicate that the value is hexadecimal.

For example, if the current language were C, you could type the hexadecimal selector value 7F as `0x7F`. For Pascal, you'd type it as `$7F`. You could also type the decimal value `127` in order to go to selector 7F.

Another method of entering the selector value is to display the CPU window and check the segment register values. If a register holds the selector you're interested in, you can enter the name of the register preceded by an underscore (_). For example, you could type the data segment register as _DS.

Examine Displays the contents of the memory area referenced by the current selector and switches focus to the pane where the contents are displayed. If the selector points to a code segment, the contents are displayed in the Code pane. If the contents are data, they're displayed in the Data pane.

The Data pane

This pane shows a raw display of an area of memory you've selected. The leftmost part of each line shows the address of the data displayed in that line. The address is displayed either as a hex segment and offset, or with the segment value replaced with one of the register names if the segment value is the same as that register. The Data pane matches registers in the following order: DS, ES, SS, CS.

Next, the raw display of one or more data items is displayed. The format of this area depends on the display format selected with the Display As local menu command. If you choose one of the floating-point display formats (Comp, Float, Real, Double, Extended), a single floating-point number is displayed on each line. Byte format displays 8 bytes per line, Word format displays 4 words per line, and Long format displays 2 long words per line.

When the data is displayed as bytes, the rightmost part of each line shows the display characters that correspond to the data bytes displayed. TDW displays all byte values as their display equivalents, so don't be surprised if you see funny symbols displayed to the right of the hex dump area—these are just the display equivalents of the hex byte values.

There are two ways of choosing an address:

- Use the local menu Goto, Follow, or Previous command.
- Position on a data selector in the Selector pane, then choose Examine to display the contents of the selector in the Data pane.

The Data pane local menu lets you go to a new address, search for a character string, change bytes at the current cursor location, follow near or far pointer chains, restore a previous address, change how data appears in the window, and move, change, read, and write blocks of memory.

The Stack pane

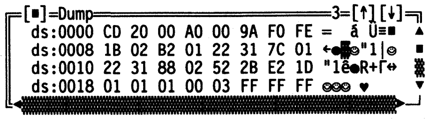
An arrow (▸) shows the current stack pointer (SS:IP).

The Stack pane, in the lower right corner of the CPU window, shows the contents of the stack. The commands in the local menu let you change positions in the stack and change values of words on the stack.

The Dump window

The Dump window, opened by choosing View | Dump, shows you a raw data dump of any area of memory. It works much like the Data pane in the CPU window (see page 183), except that, when zoomed to full size, the Dump Window shows twice as much data on a single line.

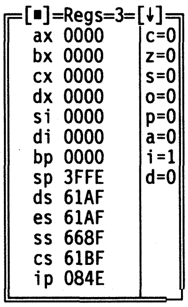
Figure 12.2
The Dump window



```
[■]=Dump                               3=[↑][↓]
 ds:0000 CD 20 00 A0 00 9A F0 FE =  á Û≡■    ▲
 ds:0008 1B 02 B2 01 22 31 7C 01 ←●▓☺"1|☺    ■
 ds:0010 22 31 88 02 52 2B E2 1D "1ê●R+Γ↔    ▓
 ds:0018 01 01 01 00 03 FF FF FF ☺☺☺ ♥       ▼
```

Typically, you use this window if you're in an Inspector window and you want to look at the raw bytes that make up the object you are inspecting. Using View | Dump from the Inspector window gets you a Dump window that's positioned to the data in the Inspector window.

The Registers window



```
[■]=Regs=3=[↓]
 ax 0000   c=0
 bx 0000   z=0
 cx 0000   s=0
 dx 0000   o=0
 si 0000   p=0
 di 0000   a=0
 bp 0000   i=1
 sp 3FFE   d=0
 ds 61AF
 es 61AF
 ss 668F
 cs 61BF
 ip 084E
```

The Registers window shows you the contents of the CPU registers and flags. It works like a combination of the Registers and Flags panes in the CPU window (see page 181).

You can perform the same functions from the local menu of the Registers window as you can from the local menus of the Registers pane and the Flags pane.

C H A P T E R

13

Debugging a standard Pascal application

Debugging is like the other phases of designing and implementing a program—part science and part art. There are specific procedures that you can use to track down a problem, but at the same time, a little intuition goes a long way toward making a long job shorter.

The more programs you debug, the better you get at rapidly locating the source of problems in your code. You learn techniques that suit you well, and you unlearn methods that have caused you problems.

In this chapter, we discuss some different approaches to debugging, talk over the different types of bugs you may find in your programs, and suggest some ways to test your program to make sure that it works—and keeps on working.

Let's begin by looking at where to start when you have a program that doesn't work correctly.

When things don't work

First and foremost, don't panic! Even the most expert programmer seldom writes a program that works the first time.

To avoid wasting a lot of time on fruitless searches, try to resist the temptation to randomly guess where a bug might be. It is better to use a universally tried-and-true approach: divide and conquer.

Make a series of assumptions, testing each one in turn. For example, you can say, "The bug must be occurring before function *xyz* is called," and then test your assumption by stopping your program at the call to *xyz* to see if there's a problem. If you do discover a problem at this point, you can make a new assumption that the problem occurs even earlier in your program.

If, on the other hand, everything looks fine at function *xyz*, your initial assumption was wrong. You must now modify that assumption to "The bug is occurring sometime *after* function *xyz* is called." By performing a series of tests like this, you can soon find the area of code that is causing the problem.

That's all very well, you say, but how do I determine whether my program is behaving correctly when I stop it to take a look? One of the best ways of checking your program's behavior is to examine the values of program variables and data structures. For example, if you have a routine that clears an array, you can check its operation by stopping the program after the function has executed, and then examining each member of the array to make sure that it's cleared.

Debugging style

Everyone has their own style of writing a program, and everyone develops their own style of debugging. The debugging suggestions we give here are just starting points that you can build on to mold your own personal approach.

Many times, the intended use of a program influences the approach you take to debug it. If a program is for your own use or will only be used once or twice to perform a specific task, a full-scale testing of all its components is probably a waste of time, particularly if you can determine that it is working correctly by inspecting its output. If a program is to be distributed to other people or performs a task of which the accuracy is hard to determine by inspection, your testing must be far more rigorous.

Run the whole thing

For a simple or throwaway program, the best approach is often just to run it and see what happens. If your test case has problems, run the program with the simplest possible input and check the output. You can then move on to testing more complicated input cases until the output is wrong. This testing procedure will give you a good feeling for just how much or how little of the program is working.

Incremental testing

When you want to be very sure that a program is healthy, you must test the individual routines, as well as checking that the program works as expected for some test input data. You can do this in a couple of ways: You can test each routine as you write it by making it part of a test program that calls it with test data. Or you can use TDW to step through the execution of each routine when the whole program is finished.

Types of bugs

Bugs fall into two broad categories: those peculiar to the language you're working in and those that are common to any programming language or environment.

By making mental notes as you debug your programs, you learn both the language-specific constructs you have trouble with, and also the more general programming errors you make. You can then use this knowledge to avoid making the same mistakes in the future, and to give you a good starting point for debugging future programs.

Understanding that each bug is an instance of a general family of bugs or misunderstandings will improve your ability to write errorless code. After all, it's better to write bug-free code than to be really good at finding bugs.

General bugs

The following examples barely scratch the surface of the kinds of problems you can encounter in your programs.

Hidden effects

If you are careless about using global variables in procedures or functions, a call to a procedure or function can leave unexpected contents in a variable or data structure:

```
program Buffers;

uses WinCrt, Strings;

var
  WorkBuf,
  AllCaps: PChar;

procedure Convert(S: PChar);
var
  I, Len: Integer;

begin
  Len := StrLen(s);
  StrCopy(WorkBuf, S);
  for I := 1 to Len do
    {...} ;
end;


begin
  WorkBuf := 'all done';
  AllCaps := 'SNAFU';
  Convert(AllCaps);
  Writeln(WorkBuf);
end.
```

Here, the correct thing to do would be to have the procedure use its own private work buffer.

Assuming initialized data

Don't assume that another routine has already set a variable for you:

```
uses Strings;

var
  WorkBuf: PChar;

procedure AddWorkString(S: PChar);
```

```
begin
  StrCopy(WorkBuf, S); { oops }
end;
```

You should code a routine of this sort defensively by adding the statement

```
if (WorkBuf = nil) then getmem(WorkBuf, 100);
```

Not cleaning up

This sort of bug can crash your program by exhausting heap space:

```
function CrunchString(S: PChar): PChar;
var
  Work: PChar
begin
  getmem(Work, 100);
  StrCopy(S, Work);
  {...}
  CrunchString := S;      { whoops--Work still allocated }
end;
```

Fencepost errors

These bugs are named after the old brain teaser that goes "If I want to put up a 100-foot fence with posts every 10 feet, how many fenceposts do I need?" A quick but wrong answer is ten (what about the final post at the far end?). Here's a simple example:

```
i := 1;
while i < 10 do
begin
  ...                      { oops--only 9, not 10 }
end;
```

Here you can easily see the numbers 1 and 10, and you think that your final test value is ten. (Better make that **<** into a **<=**.)

Pascal-specific bugs

Because of the strong type-checking and error-checking features of Pascal, there are few bugs endemic to the language itself. However, because Turbo Pascal for Windows gives you the power to turn off much of that error checking, you can introduce errors that you might not have otherwise. And even with Pascal, there are ways of getting into trouble.

Uninitialized variables

Turbo Pascal does not initialize variables for you; you must do it yourself, either through assignment statements or by declaring them as typed constants. Consider the following program:

```
program Test;

uses WinCrt;

var
  I,J,Count: Integer;
begin
  for I := 1 to Count do begin
    J := I*I;
    Writeln(I:2, ' ', J:4)
  end
end.
```

Count has whatever random value occupied its location in memory when it was created, so you have no idea how many times this loop is going to execute.

Furthermore, variables declared within a procedure or function are created each time you enter that routine and destroyed when you exit; you cannot count on those variables retaining their values between calls to that routine.

Problems with pointers

Pointer errors can be the most elusive of bugs. When a program hangs or produces strange, unpredictable results, one of the first places to look is at your use of pointers.

Three common errors occur with pointers. The first is using them before you have assigned them a value (**nil** or otherwise). Just like any other variable or data structure, a pointer is not automatically initialized just by being declared. It should be explicitly set to an initial value (by passing it to *New* or assigning it **nil**) as soon as possible.

Second, don't reference a **nil** pointer, that is, don't try to access the data type or structure that the pointer points to if the pointer itself is **nil**. For example, suppose you have a linear linked list of records, and you want to search it for a record with a given value. Your code might look like this:

```
function FindNode(Head: NodePtr; KeyVal: Integer): NodePtr;
var
  Temp : NodePtr;
begin
  Temp := Head;
  while (Temp^.KeyVal <> Val) and (Temp <> nil) do
    Temp := Temp^.Next;
  FindNode := Temp
end; { of function FindNode }
```

If *Val* isn't equal to the *Key* field in any of the nodes in the linked list, this code tries to evaluate `Temp^.Key` when *Temp* is **nil**, resulting in unpredictable behavior. Solution? Rewrite the expression to read

```
while (Temp <> nil) and (Temp^.Key <> Val)
```

In addition, enable short-circuit Boolean evaluation by using the TPW **{$B-}** option or choosing Options | Compiler to display the Compiler Options dialog box, then clicking Short Circuit. That way, if *Temp* is **nil**, the second term is never evaluated.

Finally, don't assume that a pointer is set to **nil** just because you've passed it to *Dispose* or *FreeMem*. The pointer still has its original value; however, the memory it points to is now free to be used for other dynamic variables. You should explicitly set a pointer to **nil** after disposing of its data structure.

Scope confusion

Pascal lets you nest procedures and functions very deeply, and each of those procedures and functions can have its own declarations. Consider the following program:

```
program Confused;

uses WinCrt;

var
  A,B,T: Integer;

procedure Swap(var A,B: Integer);
var
  T: Integer;
begin
  Writeln('2: A, B, T = ', A:3, B:3, ' ', T);
  T := A;
  A := B;
  B := T;
  Writeln('3: A, B, T = ', A:3, B:3, ' ', T)
end;  { of procedure Swap }
```

```
begin { main body of Confused }
  A := 10; B := 20; T := 30;
  Writeln('1: A, B, T = ', A:3, B:3, ' ', T);
  Swap(B,A);
  Writeln('4: A, B, T = ', A:3, B:3, ' ', T);
end.  { of program Confused }
```

What's the output of this program? It will look something like this:

```
1: A,B,T =  10 20 30
2: A,B,T =  20 10 22161
3: A,B,T =  10 20 20
4: A,B,T =  20 10 30
```

What's happening here is that you have two versions each of *A*, *B*, and *T*. The global versions are used in the main body of the program, while *Swap* has versions local to itself—its formal parameters *A* and *B*, and its local variable *T*. To further confuse things, we made the call *Swap(B,A)*, which means that the formal parameter *A* is actually the global variable *B* and vice versa. And, of course, there is no correlation between the local and global versions of *T*.

There was no real "bug" here, but problems can arise when you think that you're modifying something that you aren't. For example, the variable *T* in the main body didn't get changed, even though you thought it might have. This is the opposite of the "hidden effects" bug mentioned on page 188.

If you also had the following record declaration, things could get even more confusing:

```
type
  RecType = record
    A,B : Integer;
  end;

var
  A,B : Integer;
  Rec : RecType;
```

Inside a **with** statement, a reference to *A* or *B* would reference the *fields*, not the *variables*.

Superfluous semicolons

Pascal allows a "null" statement (one consisting only of a semicolon). Placed at the wrong spot, this statement can create all kinds of problems. Consider the following program:

```
program Test;

uses WinCrt;

var
  I,J : Integer;
begin
  for I := 1 to 20 do;
  begin
    J := I * I;
    Writeln(I:2,' ',J:4)
  end;
  Writeln('All done!')
end.
```

The output of this program is not a list of the first 20 integers and their squares; it's simply

```
20 400
All done!
```

That's because the statement `for I := 1 to 20 do;` ends with a semicolon. This means it executes the null statement 20 times. After that, the statements in the **begin..end** block are executed, then the final *Writeln* statement. To fix this, just eliminate the semicolon following the **do** keyword.

Undefined function return value

If you write a function, you must be sure that the function name has some value assigned to it before you exit the function. Consider the following section of code:

```
const
  NLMax = 100;
type
  NumList = array[1..NLMax] of Integer;
  ...
function FindMax(List : NumList; Count : Integer) : Integer;
var
  I, Max : Integer;
```

```
begin
  Max := List[1];
  for I := 2 to Count do
    if List[I] > Max then
    begin
      Max := List[I];
      FindMax := Max
    end
end; { of function FindMax }
```

This function works fine—as long as the highest value in *List* isn't in *List*[1]. In that case, *FindMax* never gets assigned a value. A correct version of the function would use this:

```
begin
  Max := List[1];
  for I := 2 to Count do
    if List[I] > Max then
      Max := List[I];
  FindMax := Max
end; { of function FindMax }
```

Decrementing Word or Byte variables

Be careful not to decrement an unsigned scalar (Byte or Word) while testing for >= 0. The following code produces an infinite loop:

You can use arithmetic overflow checking {$Q+} to catch this bug at runtime.

```
var
  w: Word;
begin
  w := 5;
  while w >= 0 do
    w := w - 1;
end.
```

After the fifth iteration, *w* equals 0. The next time through, it's decremented to 65,535 (because words range from 0 to 65,535), which is still >= 0. You should use an Integer or Longint in such cases.

Ignoring boundary or special cases

Note that both versions of the function *FindMax* in the previous section "Undefined function return value" assume that *Count* >= 1. However, there may be times when *Count* = 0; that is, the list is empty. If you call *FindMax* in that situation, it returns whatever happens to be in *List*[1]. Likewise, if *Count* > *NLMax*, you'll end up either generating a run-time error (if range-checking is enabled) or searching through memory locations not contained in *List* for the maximum value.

There are two possible solutions to this. One, of course, is never to call *FindMax* unless *Count* is in the range 1..*NLMax*. This isn't an impertinent comment; a serious part of good software design is to define the requirements for calling a given routine, then ensuring they are met each time that routine is called.

The other solution is to test *Count* and return some predetermined value if it isn't in the range 1..*NLMax*. For example, you might rewrite the body of *FindMax* to look like this:

```
begin
  if (Count < 1) or (Count > NLMax) then
    Max := -32768
  else
  begin
    Max := List[1];
    for I := 2 to Count do
      if List[I] > Max then
        Max := List[I]
  end;
  FindMax := Max
end;  { of function FindMax }
```

This leads to the next type of Pascal pitfall: range errors.

Range errors

Turbo Pascal has range-checking turned off by default. This produces faster, more compact code, but it also lets you commit certain types of errors, such as assigning to variables values outside their allowed range or indexing nonexistent elements in arrays as shown in the previous example.

The first step in finding such errors is to turn range-checking back on by inserting the **{$R+}** compiler option into your program, compiling the program, and running it again. If you know (or suspect) where the error is, you can put this directive above that section and add a corresponding {$*R*-} directive afterward, thus enabling range-checking for that section only. If a range error does occur, your program stops with a run-time error, and Turbo Pascal shows you where the error occurred.

☞ If you are using TPCW to compile your program, you can use the **/F** command-line option to find the error. See the *Turbo Pascal for Windows User's Guide* for more information.

One common type of range error happens when you are indexing through an array using a **while** or **repeat** loop. For example, suppose you are looking for an array element containing a certain value. You want to stop when you've found it or when you reach the end of the array. If you've found it, you want to return the index of the element; otherwise, you want to return 0. Your first effort might look like this:

```
function FindVal(List : NumList; Count,Val : Integer) : Integer;
var
  I : Integer;
begin
  FindVal := 0;
  I := 1;
  while (I <= Count) and (List[I] <> Val) do
    Inc(I);

  if I <= Count then
    FindVal := I
end; { of function FindVal }
```

Unfortunately, this approach could result in a run-time error if *Val* isn't in *List*, and you're using normal Boolean evaluation. Why? Because the last time the test is made at the top of the **while** loop, *I* equals *Count*+1. If *Count* = *NLMax*, you're beyond the limits for *List*.

The solution to this type of problem is to select short-circuit Boolean evaluation, either by by using the Turbo Pascal {**$B-**} option or choosing Options | Compiler to display the Compiler Options dialog box, then clicking Short Circuit. That way, if *I* > *Count*, the expression

```
List[I] <> Val
```

is never evaluated.

Accuracy testing

Making a program work with valid input is only part of the job of testing. The following sections discuss some important test cases that any program or routine should be subjected to before you can consider it bug free.

Testing boundary conditions

Once you think a routine works with a range of data values, you should subject it to data at the limits of the range of valid input. For example, if you have a routine to display a list from 1 to 20 items long, you should make sure it behaves correctly both when there is exactly 1 item and exactly 20 items in the list. This can flush out the one-too-few and one-too-many "fencepost" errors (described on page 189).

Invalid data input

Once you are sure that a routine works with a full range of valid input, check that it behaves correctly when it's given invalid input. Check that erroneous input is rejected, even when it's very close to valid data. For example, the previous routine that accepted values from 1 to 20 should make sure that 0 and 21 are rejected.

Empty data input

Empty data input is a frequently overlooked area, both in testing and in designing a program. If you write a program to have reasonable default behavior when some input is omitted, you greatly enhance its ease of use.

Debugging as part of program design

When you first start designing your program, you can plan for the debugging phase. One of the most basic tradeoffs in program design involves the degree to which the different parts of your program check that they are getting valid input and that their output is reasonable.

If you do a lot of checking, you end up with a very resilient program that can often tell you about an error condition but continues to run after performing some reasonable recovery. You also end up with a larger and slower program. This type of program can be fairly easy to debug because the routines themselves inform you of invalid data before the dangers can be propagated.

You can also implement a program whose routines do little or no validation of input or output data. Your program will be smaller and faster, but bad input data or a small bug can bring things to a grinding halt. This type of program can be the most difficult to debug, since a small problem can end up manifesting itself much later during execution. This makes it hard to track down the original error.

Most programs end up being a mixture of these two techniques. You should treat input from external sources (such as the user or a disk file) with greater suspicion than data from one internal routine calling another.

The sample debugging session

This sample session uses some of the techniques we talked about in the previous sections. The program you are debugging, TDDEMOWB, is a version of the demonstration program used in Chapter 3 (TDDEMOW.PAS), except this one has some deliberate bugs in it. As with TDDEMOW, TDDEMOWB uses WinCrt to display its output through Windows.

Make sure that your working directory contains the two files needed for the debugging demonstration, TDDEMOWB.PAS and TDDEMOWB.EXE. (The *B* in these file names stands for "buggy.")

Looking for errors

Before we start the debugging session, let's run the buggy demo program to see what's wrong with it. The program is already compiled and on your distribution disk.

To simplify viewing the source code and running and debugging the program, start Turbo Pascal for Windows (pick its icon from the Windows Program manager) and use File | Open to load in TDDEMOWB.

When you see the program's source code come up in its edit window, either choose Run | Run or press *Ctrl-F9* to run the program.

You see a window come up with the program's file name as its title and a prompt for lines of text. Before continuing, enlarge the window to its maximum size so you can see all the program output. Then enter two lines of text exactly as follows:

```
ABC DEF GHI
abc def ghi
```

Press *Enter* on an empty line to end your input. TDDEMOWB then prints out its analysis of your input, as follows:

```
9 char(s) in 3 word(s) in 2 line(s)
Average of 0.67 words per line

Word length:  1   2   3   4   5   6   7   8   9  10
Frequency:    0   0   3   0   0   0   0   0   0   0

Letter:        M
Frequency:     1   1   1   1   1   1   1   1   1   0   0   0   0
Word starts:   1   0   0   1   0   0   1   0   0   0   0   0   0

Letter:        Z
Frequency:     0   0   0   0   0   0   0   0   0   0   0   0   0
Word starts:   0   0   0   0   0   0   0   0   0   0   0   0   0
```

There are four separate problems with this output:

1. The number of words is wrong (3 instead of 6).
2. The number of words per line is wrong (0.67 instead of 3.00).
3. The column headings for the second and third tables display only one letter each (instead of A..M and N..Z).
4. You typed two lines, each containing a letter from A..I, but the letter frequency tables show only a count of one each for those letters.

At this point, you can close the window and return to Turbo Pascal for Windows.

Deciding your plan of attack

Your first task is to decide which problem to attack first. A good rule of thumb is to start with the problem that appears to be happening first. In this program, after procedure *Init* is called to initialize data, keyboard input is read by function *GetLine* and then processed by procedure *ProcessLine* until the user enters an empty string. *ProcessLine* scans each input string and updates the global counters. Then, the results are displayed by procedure *ShowResults*.

The average number of words per line is computed by *ShowResults*, using the number of lines and words. Since the word count seems to be off, take a look at *ProcessLine* to see how *NumWords* is updated. Even though *NumWords* is wrong, the 0.67 words-per-line figure doesn't make sense. There's probably an error in the *ShowResults* calculation, which needs your attention as well.

The column titles for all the tables are drawn at the request of *ShowResults*. You should wait until the main loop terminates before tracking down the second and third bugs. Since the letter and word counts are wrong, it's a good bet that something is amiss inside *ProcessLine*, and that's where you should start looking for the first and fourth bugs.

Now is the time to actually start debugging—after you've thought about the problem for a moment and decided on a rough plan of attack.

Starting TDW

To start the debugging sample session, make sure the Turbo Pascal edit window with TDDEMOWB is current, then use the Run | Debugger command.

TDW loads the buggy demo program and displays the startup screen. If you want to exit from the tutorial session and return to Turbo Pascal, press *Alt-X* at any time. If you get hopelessly lost, you can always reload the demonstration program and start from the beginning again by pressing *Ctrl-F2*. (Note that this doesn't clear breakpoints or watches.)

There are two approaches to debugging a routine like *ProcessLine*: Either step through it line-by-line as it executes and make sure it does the right thing, or stop the program immediately after *ProcessLine* has done its stuff and see if it did the right thing. Since both the letter and word counts are wrong, you probably ought to look inside *ProcessLine* carefully and see how characters are processed.

Moving through the program

Now you're going to run the program and step inside the call to *ProcessLine*. There are many ways to do that. You can press *F8* four times (to step over procedure and function calls), then press *F7* once (to trace into the call to *ProcessLine*). You can also move the cursor down to line 203, press *F4* (Go to Cursor command), type some letters and press *Enter*, then press *F7* once to step into *ProcessLine*.

There are even more ways to get into *ProcessLine*. Try this one: Press *Alt-F9*. A dialog box pops up, prompting you to enter a code address to run to. Type `processline` and press *Enter*. The program will now run until *ProcessLine* gains control. When you are prompted to enter a string, enter the same data as before (that is, `ABC DEF GHI`).

After you enter the first line of data, TDW returns you to the Module window, which is displaying the first line of *ProcessLine*. *ProcessLine* contains several loops. An outer one scans the entire string. Inside that loop, there's one loop to skip over non-letters, and a second one to process words and letters. Move the cursor to the **while** loop on line 159 and press *F4* (Go to Cursor).

This loop keeps scanning until it reaches the end of the string or until it finds a letter. Each character scanned is checked via a call to a Boolean function, *IsLetter*. Press *F7* to trace into *IsLetter*. *IsLetter* is a nested function that takes a character value and returns True if it's a letter; otherwise, False. A not-very-close look reveals that it checks only for uppercase letters. It should either check for characters in the range *A* to *Z* and *a* to *z*, or it should convert the character to uppercase before performing the test.

A quick look at both lines of input that you originally entered provides a further clue to the source of the bug: You entered both uppercase and lowercase letters from *A* to *I*, but only the uppercase letters entered were displayed in the totals. Now you can see why. Since the second line of input you originally entered, `abc def ghi`, contained only lowercase letters, each character was treated as whitespace and skipped. Skipping letters throws off both the letter counts and the word count and solves the mysteries of bugs #1 and #4.

Get back to the line after the one that called *IsLetter* by another navigation technique: Press *Alt-F8*, which runs past the end statement of the current procedure or function.

The Evaluate/Modify dialog box

By the way, there's another powerful way to verify *IsLetter*'s misbehavior. Invoke the Evaluate/Modify dialog box by pressing *Alt-D E*, then enter the following expression and press *Enter*:

```
IsLetter('a') = IsLetter('A')
```

A and *a* are both letters, but the evaluation False in the Result box confirms that they're not treated the same by *IsLetter*. (You can use the Evaluate/Modify dialog box and Watches window to evaluate expressions, perform assignments, or, as you did here, call procedures and functions. For more information, refer to Chapter 6.)

Press *Esc* to get rid of the Evaluate/Modify dialog box.

Inspecting

Two bugs down, two to go. Bug #2 is much easier to find than the previous ones. Press *Alt-F8* to exit *ProcessLine,* then move the cursor to line 207 and press *F4* to run to the cursor position.

TDDEMOWB prompts you for a string. Type `abc def ghi` and press *Enter,* then press *Enter* the second time the prompt appears. Now press *F7* to step into *ShowResults.*

Remember, you're trying to find out why the average number of words per line is incorrect. The first line in *ShowResults* calculates the number of lines per word instead of words per line. Clearly, those two terms should be reversed.

As long as you're here, you might as well make sure that *NumLines* and *NumWords* have the values you'd expect. *NumLines* should equal 2, and—because of the *IsLetter* bug you've uncovered but haven't fixed—*NumWords* should equal 3. Move the cursor to *NumLines* and press *Alt-F10 I* to inspect a variable. The Inspector window shows you *NumLines'* address, type, and current value in both decimal and hexadecimal. The value is indeed equal to 2, so you can move on and have a look at *NumWords.* Press *Esc* to close the Inspector window, move the cursor forward to *NumWords,* and press *Alt-F10 I* again, or use the *Ctrl-I* hot key. *NumWords* has the expected (incorrect) value of 3, so you can move on.

Or can you? There's another problem with this calculation, and it's not even on our list. There is no check to see whether the second term is 0 before the division is performed. If you run the program from the beginning and enter no data at all (just press *Enter* when prompted), the program crashes even after you reverse the divisor and the dividend.

To confirm this, press *Esc* to close the Inspector window and type *Alt-R P* (or *Ctrl-F2*) to end the current debug session and reload the program. Then press *F9* to run the program from the beginning, and press *Enter* at TDDEMOWB's string prompt. The program terminates with runtime error 200. Press *Enter* to return to TDW, and then *Esc* to get rid of the "Program terminated" message.

Now that you know what to test for, you should modify the statement at line 95 to read

```
if NumLines <> 0 then
  AvgWords := NumWords / NumLines
else
  AvgWords := 0;
```

So much for bugs #2 and #2a. As long as you're tinkering with the Inspector window, try using it to "walk" through a data structure. Move the cursor up to the declaration of *LetterTable* on line 50. Place the cursor on the word *LetterTable*, and press *Ctrl-I*. You can see it's an array of records, 26 elements long. Use the cursor keys to scroll through each element of the array, and press *Enter* to step into one of the array elements.

Watches

You've still got to squash that column title bug (#3) in *ShowResults*. Since you already terminated the program when you tracked the divide-by-zero error, prepare for another session by closing the Inspector window, then pressing *Alt-R P* (to reset the program). Next, press *Alt-F9*, type `showresults`, and press *Enter*. Now type the all-too-familiar data `ABC DEF GHI` and press *Enter* again. Finally, type `abc def ghi` and press *Enter* twice. TDW should be stopped at *ShowResults*.

ShowResults uses a nested procedure, *ShowLetterInfo*, to display the letter tables. Move the cursor down to line 113, press *F4*, then press *F7* to step into *ShowLetterInfo*.

There are three **for** loops. The first one displays the column titles, and the second and third display frequency counts. Use *F7* to step to the first loop on line 70. Position the cursor over *FromLet* and *ToLet* and use *Ctrl-I* to check their values. They look okay (the first equals *A*, and the second equals *M*). Press *Alt-F5* to view the User screen and see where things stand. Press any key to return to the Module window.

When you're stepping through a loop like this, the Watches window is very handy; position the cursor over `ch` and press *Ctrl-W*. Now use *F7* to step through the **for** loop. As expected, it steps down to the *Write* statement on line 71. If you look at the Watches window, though, you'll see that *ch*'s value is already *M*. (It already executed the entire loop!) There's an extra semicolon right after the keyword **do**, making the **for** loop do absolutely nothing 13 times. When control falls through to the *Write* statement on line 71, the current value of *ch*, *M*, is output and the program moves on. Removing that extra semicolon eliminates bug #3.

The end

Whew. That's all the (known) bugs in this program. Perhaps you'll find some more as you step through the code. You can fix the bugs (they are marked with two asterisks (**) for your convenience) and then recompile; or you can run TDDEMOW.PAS, the bug-free version of this program, discussed in Chapter 3.

C H A P T E R

14

Debugging an ObjectWindows application

The sample Windows programs in this chapter were written using the ObjectWindows classes that make Windows programming so much easier than programming using only the Windows API.

The programs are TDODEMO and TDODEMOB (the *B* stands for buggy). TDODEMOB has several bugs in it that you will discover by working through this chapter.

Before continuing, it might be helpful if you start TDODEMO from Windows and play with it a bit to get an idea of how it works. You can either use the Program Manager File | Run command to start TDODEMO.EXE or add it to a program group as an icon.

About the program

TDODEMO is an ObjectWindows program that lets you use a mouse to scribble in various colors on the screen. When you click the left mouse button and drag the mouse, the program draws on the screen. You can clear the window by clicking the right mouse button. TDODEMO has a menu bar that lets you pick any of four pen colors: Red, Green, Blue, or Black.

You draw by pressing the mouse button, moving the mouse, and releasing the mouse button. The program accomplishes this task easily by using the ObjectWindows library and *dynamic virtual methods*. A dynamic virtual method is a virtual method with a numeric identifier attached to it.

Because Turbo Pascal for Windows defines Windows message names as numeric constants, you can use a Windows message name as the identifier of a dynamic method. ObjectWindows can then call the method whenever the window for which the method is declared receives a message that matches the method's identifier. If there is no method with an identifier matching the Windows message, ObjectWindows calls the default window procedure.

For example, in order to create a method that responds to WM_MOUSEMOVE messages, you can define a method within a window object that looks like this:

```
procedure WMMouseMove(var Msg: TMessage); virtual WM_MOUSEMOVE;
```

As you can see, you attach the identifier WM_MOUSEMOVE to the procedure by using the **virtual** <*identifier*> statement immediately after the procedure declaration.

The type *TMessage* contains the Windows window procedure parameters *wParam* and *lParam*. These parameters often hold additional information about the message, such as where the mouse is positioned.

The next few sections explain how the TDODEMOB program works. They purposely gloss over the bugs so you can discover them later. It might be helpful to start Turbo Pascal for Windows and open TDODEMOB.PAS so you can follow along in the code.

The Scribble window type definition

The Scribble window type is defined as follows:

```
type
  ScribbleWindow = object(TWindow)
    HandleDC: HDC;
    ButtonDown: Boolean;    { Left button down flag. }

    constructor Init(aParent: PWindowsObject; aTitle: PChar);
    procedure WMLButtonDown(var Msg: TMessage);
      virtual WM_LBUTTONDOWN;
    procedure WMLButtonUp(var Msg: TMessage);
      virtual WM_LBUTTONUP;
    procedure WMMouseMove(var Msg: TMessage);
      virtual WM_MOUSEMOVE;
    procedure WMRButtonDown(var Msg: TMessage);
      virtual WM_RBUTTONDOWN;
  end;
```

The *ScribbleWindow* type defines a window object that responds to the following user input:

- Mouse movements
- Left mouse button press and release
- Right mouse button press

There are two instance variables, *HandleDC* and *ButtonDown*, that hold a device class and the state of the mouse button, respectively.

Init

The constructor *Init* calls the standard constructor *TWindow.Init*, which allows the window to behave like any other *TWindow*, and also initializes the *ButtonDown* variable to false.

WMLButtonDown

When the user presses the mouse button in the Scribble window and is about to draw, the window receives a WM_LBUTTONDOWN message, which causes ObjectWindows to call *WMLButtonDown* (since it has an identifier of WM_LBUTTONDOWN). *WMLButtonDown* moves the pen to the current position of the mouse and sets the *ButtonDown* variable to indicate that the button is down. There are additional Windows calls this procedure should be making that will be discussed later.

WMMouseMove

Once the user starts moving the mouse over the window, the window begins receiving WM_MOUSEMOVE messages, which cause ObjectWindows to call the method *WMMouseMove*. If the user has pressed the left mouse button, the program draws a line each time the mouse is moved. If the user hasn't pressed the mouse button, nothing at all happens.

WMLButtonUp

When the user finishes scribbling and releases the mouse button, the window receives a WM_LBUTTONUP message, which in turn causes ObjectWindows to call *WMLButtonUp*. The program marks the *ButtonDown* variable False and releases the device class associated with the window.

WMRButtonDown

When the user presses the right button to clear the screen, ObjectWindows calls the method *WMRButtonDown*, which calls the Windows procedure *UpdateWindow*. Calling this procedure is supposed to clear the window.

Adding color with CScribbleWindow

The program now has an object that allows the user to scribble with a mouse. In order to add color, it would be possible to add more methods to the ScribbleWindow type, but it might be more useful to use the OOP *inheritance* feature and create a new object from *ScribbleWindow* instead. The new object is called *CScribbleWindow* and is declared as follows:

```
CScribbleWindow = object(ScribbleWindow)
  thePen: HPen;
  constructor Init(aParent: PWindowsObject; aTitle: PChar);
  destructor Done; virtual;
  procedure SelectRedPen(var Msg: TMessage);
    virtual cm_First + RedMenu;
  procedure SelectGreenPen(var Msg: TMessage);
    virtual cm_First + GreenMenu;
  procedure SelectBluePen(var Msg: TMessage);
    virtual cm_First + BlueMenu;
  procedure SelectBlackPen(var Msg: TMessage);
    virtual cm_First + BlackMenu;
  procedure WMLButtonDown(var Msg: TMessage);
    virtual WMLButtonDown;
  procedure GetWindowClass(var AWndClass: TWndClass); virtual;
  function GetClassName: PChar; virtual;
end;
```

thePen The new object has an instance variable *thePen* that stores the current pen color. Using this variable, the *CScribbleWindow* methods set the pen color and then selects it into the device class.

Init The constructor *Init* first calls its ancestor's constructor. It then attaches a menu to the window by setting the window *Attr.Menu* field to the menu handle returned by the call to *LoadMenu*. *LoadMenu* loads the menu from the resource file. Finally, the constructor initializes a pen with the color black.

Done The destructor *Done* calls its ancestor's destructor and disposes of the pen that was created.

The pen-color routines There are four methods that set the pen color by deleting the current pen and creating a new one of the correct color. These methods differ only in the color each one sets.

WMLButtonDown All that's left to do is to select the pen into the device context before any drawing takes place. Since drawing takes place after the user presses the left mouse button, it's necessary to redeclare the dynamic virtual method *WMLButtonDown* so it gets called instead of the original in *ScribbleWindow*.

The new version of *WMLButtonDown* first calls the parent object's *WMLButtonDown* method, and then selects *thePen* into the current device context.

Creating the application

To create an application that uses the Color Scribble window, it's necessary to create an object based on the ObjectWindows object *TApplication*. The purpose of this object, *CScribbleApplication*, is to redeclare the *InitMainWindow* method so that the application can create a main window with the properties of *CScribbleWindow*.

Now that you know how the program works, you can begin to debug it.

Debugging the program

If you haven't done so already, start Turbo Pascal for Windows, load TDODEMOB.PAS, then compile and run it. When you press the mouse button and move the mouse around, you'll notice that nothing gets drawn. Obviously, there's a bug.

Finding the first bug

Since lines are drawn only in *WMMouseMove*, that would be a good place to start investigating. One of three things is probably happening.

- *WMMouseMove* isn't getting called when the mouse is moved.
- There's a problem with the call to the Windows function *LineTo*.
- The *ButtonDown* variable isn't being set to True.

Load the program into TDW. You can do this most easily from within Turbo Pascal by loading TDODEMOB.PAS into an Edit window (which you've already done) and choosing Run | Debugger.

Eliminating the alternatives

The first thing to test is whether *WMMouseMove* is getting called. Move the cursor to *ScribbleWindow.WMMouseMove* and set a breakpoint (press *F2*) at the **begin** statement. Press *F9* to run the program. (You might have to press *F9* several times if the program keeps exiting to TDW.)

Once the window is up and running, move the mouse over the window without pressing the left mouse button. TDW stops execution of the program and returns you to the breakpoint you set in *WMMouseMove*. (You might have to press a key to get Windows to release the WM_MOUSEMOVE messages and make this work.) *WMMouseMove* is getting called, so that takes care of the first possible cause. The bug is either that *ButtonDown* isn't getting set or that there's a problem with the call to *LineTo*.

To test both these possibilities, remove the breakpoint from the *WMMouseMove* **begin** statement, then move the cursor down two lines to the call to the *LineTo* function and set a breakpoint there. Next, move the cursor to *ScribbleWindow.WMLButtonUp* and set another breakpoint at the **begin** statement of that routine.

When you run the program again, if you move the mouse with the left button down and execution stops at the call to *LineTo*, you'll know that *ButtonDown* was set properly and the problem is with the *LineTo* call; otherwise, the program returns to *WMLButtonUp* and the problem is with *ButtonDown*.

Resume execution of the program by pressing *F9*. With the cursor in the window, press the left mouse button and move the mouse. Since execution stops at *LineTo*, *ButtonDown* must hold the correct value. The problem must be with the call to *LineTo*.

Debugging LineTo

LineTo takes three parameters: a display context (*HandleDC*), an *x* position (*Msg.LParamLo*), and a *y* position (*Msg.LParamHi*). Windows sends the current mouse position with WM_MOUSEMOVE messages in the *lParam* field of the message. The *TMessage* variable *Msg* picks up this parameter in its own *lParam* fields, *LParamLo*, the *x* value, and *LParamHi*, the *y* value. (*TMessage* is a type provided by ObjectWindows.) To be valid, the display context must be a number greater than zero, and the *x* and *y* values must be less than the size of the window in pixels.

Move the cursor to the *HandleDC* parameter of *LineTo* and press *Ctrl-I* to inspect its value. As you can see, the value is zero, indicating that the context was never associated. Since you can't write to a context that hasn't been associated, you've probably found the problem. Press *Esc* when you're finished inspecting this value.

ScribbleWindow.WMLButtonDown is responsible for setting up the window for drawing, which should include initializing the display context for the window. If you look at the code for *WMLButtonDown*, you see that *HandleDC* was never associated with a display context. The following code shows *WMLButtonDown* with the context initialization statement added:

```
procedure ScribbleWindow.WMLButtonDown(var Msg: TMessage);
begin
  if not ButtonDown then
  begin
    ButtonDown := True;           { Mark mouse button as being     }
                                  { pressed so when mouse movement }
                                  { occurs, a line will be drawn.  }
    HandleDC := GetDC(HWindow);{Create display context for drawing.}

    MoveTo(HandleDC, Msg.LParamLo,{ Move drawing point to location }
           Msg.LParamHi);         { where mouse was pressed.       }

  end;
end;
```

Testing the fix

If you exit TDW before running and exiting the application, Windows will probably start the application anyway, and you'll have to exit it then.

Remove the breakpoints at the call to *LineTo* and at *ScribbleWindow.WMLButtonUp*. Next, run Color Scribble and exit it, then exit TDW. When you are back in Turbo Pascal, move to *ScribbleWindow.WMLButtonDown* and add the context initialization statement.

Before compiling the program, make sure debug information will be written to it by doing the following:

1. Choose Options | Compiler and check Debug Information
2. Choose Options | Linker and check Debug Info In EXE.
3. Choose Options | Save to save your settings.

Now you can recompile the program (choose Compile | Compile or press *Alt-F9*) and run it.

When you press the mouse button and move the mouse, black lines now appear. Try different colors by selecting pen colors from the menu. Red, green, and blue all work fine; however, when you try to change the pen color back to black, the pen won't change color. It looks like you've found another bug.

Finding the pen color bug

Turbo Pascal might prompt you to save all modules when you run TDW. You can click Yes to save them.

The most likely culprit for this bug is the *CScribbleWindow* method that creates a black pen, *SelectBlackPen*. Exit Color Scribble, then start TDW again and set a breakpoint at the **begin** statement of *CScribbleWindow.SelectBlackPen*. Then run the program and choose Pen | Black. TDW should have stopped execution at the breakpoint. Since it didn't, something else must be wrong.

It appears that *SelectBlackPen* is never being called. Because this routine relies on the dynamic virtual method table to get called, it's possible that there's something wrong with the identifier.

Setting a window breakpoint

When a user chooses a menu item, Windows sends a WM_COMMAND message to the window that owns the menu. The *wParam* parameter of the message contains the identifier of the menu item that was selected. When an ObjectWindows window receives a WM_COMMAND message, it scans through the dynamic method indexes of the window object looking for the value *cm_First+menu_id*. *SelectBlackPen* has an index of *cm_First+BlackMenu*, where *BlackMenu* has the value 104.

In order to find out what the *wParam* parameter of the Pen | Black command message is, you need to tell TDW to stop execution when it receives a WM_COMMAND message. You can then run the program, make the menu selection, and then check *wParam* to see if it matches the constant *BlackMenu*.

Interrupting the program with Ctrl-Alt-SysRq

See the next section if Ctrl-Alt-SysRq doesn't work on your system.

Before you can set the program to break on receipt of the WM_COMMAND message, you must first interrupt the program and give control back to TDW. You do that by pressing *Ctrl-Alt-SysRq*. TDW returns you to the CPU window, indicating that Windows kernel code was executing when the program exited.

Since you aren't in the TDOEMOB module window, you must use a roundabout method to set the message breakpoint. Press *Alt-F3* to remove the CPU window, then press *Ctrl-F4* to bring up the Evaluate/modify window. Enter the window handle name *CSApp.MainWindow^.hWindow* as the expression to evaluate, then press *Enter*. A numeric value, the value of the window handle, appears in the result box. Write this value down, then press *Esc* to close the Evaluate/modify window.

Setting the breakpoint

Next, choose View | Windows Messages, which displays the Windows Messages window. The upper left pane shows what windows you are currently monitoring. The upper right pane shows what types of messages you are interested in. The bottom pane contains messages that have been received. Currently, all three panes are blank because you haven't added any windows yet.

Programs written using ObjectWindows have no easily accessed window procedure, so it's best to use the window handle instead.

To add a window, normally you would move the cursor to the upper left pane, then begin typing the name of the window handle. Since the menu is attached to the main window, you want to log messages coming into that window. (It's the only window in the program, which simplifies things.) The window handle of the main window is *CSApp.MainWindow^.hWindow*.

However, because the TDODEMOB module window isn't active, you must type instead the value of the window handle you retrieved earlier, then press *Alt-H* to tell TDW that this value represents a handle and not a procedure. You'll see the Identify By radio button move to Handle. Finally, press *Enter* or select OK to add the handle value to the pane.

You should only select a window handle after it's been set in the program. A handle is set after the window object is initialized. In this program, the handle is set after the method *CScribbleWindow.SetupWindow* has executed.

The next step is to specify the WM_COMMAND message as the one to break on. Move the cursor to the upper right pane, then press *Alt-F10* to bring up the local menu. Choose Add, type `WM_COMMAND`, press *Alt-B* to push the Break button, then press *Enter*. The upper right pane now displays `Break on message WM_COMMAND`.

If Ctrl-Alt-SysRq doesn't work

Ctrl-Alt-SysRq doesn't work on all systems. If it doesn't work on yours, you have to terminate Color Scribble, then use *Ctrl-F2* to reload TDODEMOB. When the module window comes up, the next step is to set a breakpoint in the code so you can run the program, initialize the window handle, then exit back to TDW and set the message breakpoint.

To set the breakpoint, in the Module window press *Ctrl-PgUp* to go to the beginning of the file. You'll be setting the breakpoint on the *ScribbleWindow* method *WMLButtonDown*, so press *Ctrl-S* to bring up the Search dialog box, type `ScribbleWindow.WMLButtonDown`, then press *Enter*. When the cursor is on the method name, press *F2* to set a breakpoint on the *begin* statement.

The next step is to press *F9* to run the program. When the Color Scribble window comes up, press the left mouse button to break and return to TDW. Remove the breakpoint by pressing *F2*.

Now you're ready to set the message breakpoint. Go back to the previous section and begin reading at *Setting the breakpoint*. Since the Module window is active and *CSApp.MainWindow^.hWindow* is set, you can enter the handle name instead of its numeric value. Otherwise, the process is the same.

Inspecting wParam

You can now resume execution of the program by pressing *F9*.

Choose Pen | Black Pen from the menu. Once you have selected a black pen, TDW stops execution and displays the CPU Window, indicating that the program was not executing your code at the time the break occurred. Close the CPU window by pressing *Alt-F3*.

If necessary, bring up the Windows Messages window again. Zoom the window to full size so you can see the entire message in the lower pane. You can see that the window received a WM_COMMAND message with 204 ($CC) in the *wParam* parameter. But the constant *BlackMenu* is 104, not 204. The reason the virtual method was not getting called was because the application was looking for an identifier of *cm_First+204*, but its value was actually *cm_First+104*. If you change *BlackMenu* to 204, selecting a black pen should work correctly.

When you've made this change, the constant declarations at the beginning of the program will be as follows:

```
const
  PenWidth   = 1;         { Width of Scribble line.      }
  MenuID     = 100;       { ID of menu in resource file. }
  IconID     = 100;       { ID of icon in resource file. }
  RedMenu    = 101;       { Value of Pen|Red menu.       }
  GreenMenu  = 102;       { Value of Pen|Green menu.     }
  BlueMenu   = 103;       { Value of Pen|Blue menu.      }
  BlackMenu  = 204;       { Value of Pen|Black menu.     }
```

Testing the fix

Run Color Scribble and exit it, then exit TDW. When you are back in Turbo Pascal, change the *BlackMenu* constant definition, then recompile the program and run it.

Now when you draw in the window, you might notice another problem. If, as you're drawing, you move the mouse off the window, then back onto the window at another location, you'll see that the program has drawn a straight line connecting the point where you left the window and the point where you came back on.

What the program should do is just stop drawing when you leave the window and start drawing when you come back. You've discovered yet another bug.

Finding the off-screen drawing bug

A place to start looking for this bug is in the window messages the window receives. Get out of the Color Scribble program and load TDODEMOB.PAS into TDW. Before setting any messages, you have to initialize the main window so the window handle will be valid. To do that, set a breakpoint at *CScribbleWindow.WMLButtonDown,* then press *F9* to run the program. Next press the left mouse button to return to the TDW

Module window. Once you're back in the Module window, remove the breakpoint you just set.

Logging the window messages

Bring up the Windows Messages window with View | Windows Messages and type `CSApp.MainWindow^.hWindow` to enter this window handle in the left pane of the Windows Messages window. Press *Alt-H* to select the Handle button, then press *Enter*.

Next, move the cursor to the top right pane and press *Ctrl-A* to bring up the Set Message Filter dialog box. Then set a break on WM_LBUTTONUP so that TDW regains control after you finish drawing. You want to look at all the messages that come back, but setting WM_LBUTTONUP erased the Log All Messages setting. What you must must do is once again bring up the Set Message Filter dialog box, this time selecting the Log All Messages class.

Discovering the bug

Resume execution of TDODEMOB by pressing *F9*. Begin drawing, then move the mouse off the client area and back on again at another place. To reduce the number of messages, just move off and right back on again, then release the left mouse button so control returns to TDW.

Before looking at the Window Messages window, make sure to zoom it to full size (press *F5*) so you can see more messages. When you look in the lower pane of the Windows Messages window, you see a lot of WM_NCHITEST and WM_SETCURSOR messages. Interspersed among these messages are a WM_LBUTTONDOWN message, a number of WM_MOUSEMOVE messages, followed by some WM_NCMOUSEMOVE messages, followed by more WM_MOUSEMOVE messages, and a final WM_LBUTTONUP message. It seems that when the cursor is not over the window, it doesn't receive any WM_MOUSEMOVE messages, only WM_NCMOUSEMOVE messages.

Now it becomes clear what the bug is. The program draws from the location of the last WM_MOUSEMOVE message to the location of the current WM_MOUSEMOVE message. When the mouse exits the client area, the program doesn't receive any WM_MOUSEMOVE messages. Therefore, when the mouse returns to the client area, the last location is where it left the screen, and the program obediently draws a line from that location to the current location.

Fixing the bug

One possible solution would be to determine when the mouse is off the client area so the program can ignore the last mouse position and begin drawing again when the mouse reenters the client area. That would require some fancy logic to determine when the mouse was leaving the client area of the window and when it moves back over the client area. Fortunately, there is an easier way.

The Windows function *SetCapture* does exactly what's needed. This function tells Windows to send all mouse-related messages to the specified window until the program calls *ReleaseCapture,* thus causing the window to receive WM_MOUSEMOVE messages instead of the nonclient WM_NCMOUSEMOVE messages.

If you put *SetCapture* in *ScribbleWindow.WMLButtonDown* and *ReleaseCapture* in *WMLButtonUp, WMMouseMove* will actually draw outside the window when the mouse is scribbling outside the window, but Windows will clip the drawing for the program.

These changes are shown in the following code listing:

```
procedure ScribbleWindow.WMLButtonDown(var Msg: TMessage);
begin
  if not ButtonDown then
  begin
    ButtonDown := True;           { Mark mouse button as being      }
                                  { pressed so when mouse movement }
                                  { occurs, a line will be drawn.  }

    SetCapture(HWindow);         { Tell windows to send all mouse   }
                                 { messages to window. WMLButtonUp  }
                                 { method will release the capture. }

    HandleDC := GetDC(HWindow); {Create display context for drawing.}

    MoveTo(HandleDC, Msg.LParamLo, { Move drawing point to location }
           Msg.LParamHi);          { where mouse was pressed.       }

  end;
end;
```

```
procedure ScribbleWindow.WMLButtonUp(var Msg: TMessage);
begin
  if ButtonDown then
  begin
    ReleaseCapture;        { Tell Windows to stop sending all mouse  }
                           { messages to this window.  Allow other   }
                           { applications to receive mouse messages. }
    ReleaseDC(HWindow,handleDC);{ Release display context created    }
                                { by WMLButtonDown method.           }
    ButtonDown := False;        { Mark mouse button as not pressed.  }
  end;
end;
```

Testing the fix

Run Color Scribble and exit it, then exit TDW. When you're back in Turbo Pascal, enter the changes to the two routines, then recompile the program and run it. Now when you draw on the window, everything works fine, but when you try to erase the screen by using the right mouse button, nothing happens. You've found another bug.

Finding the erase-screen bug

Exit Color Scribble, then load TDODEMOB into TDW. To execute to *WMRButtonDown,* the procedure where the bug probably is, press *Alt-F9* and type `WMRButtonDown`. Scribble a little in the window, then press the right mouse button. TDW stops the program at the beginning of *WMRButtonDown.*

Using the *F7* key, step into *WMRButtonDown* and stop at the call to *UpdateWindow.* The only parameter is *HWindow.* You can assume that *HWindow* has been set correctly because other methods are using it successfully. Since there's nothing obviously wrong, one thing you can do is test to see if the WM_PAINT message that should be sent to the window by the call to *UpdateWindow* is actually being received by the window.

By now you probably know how to set a message breakpoint on WM_PAINT. If not, review the description of how to set a message breakpoint for WM_COMMAND on page 213.

After setting the message breakpoint, press *F9* to run the program. Since it doesn't break and return, WM_PAINT is never getting sent to the window. For some reason, calling *UpdateWindow* is not working as expected.

Analyzing the cause of the bug

This bug requires a little understanding of how Windows handles the *UpdateWindow* function. When a program calls this function, Windows checks to see if any part of the window is invalid and needs repainting. If so, Windows sends a WM_PAINT message to the window. If not, there's no reason to waste system resources with an unnecessary message, so Windows does nothing. But how does Windows know that the window needs updating?

An application notifies Windows that at least part of the Window is invalid by calling either *InvalidateRect* or *InvalidateRgn*. These two functions put an update area in the window and notify Windows that it should update the window with a WM_PAINT message. However, Windows assigns a low priority to the WM_PAINT message it sends in response to either of these function calls, so if you want to ensure that the window gets updated immediately, you should retain the call to *UpdateWindow*.

Fixing the bug

Adding a call to *InvalidateRect* in *WMRButtonDown* will fix the problem. *InvalidateRect* takes three parameters, a window handle that identifies the window, a pointer to a rectangle that marks the rectangle to be added to the update rectangle, and a Boolean parameter that specifies if the rectangle should be erased. You can pass in *nil* for the pointer to the rectangle, telling Windows that the entire window should be added to the update rectangle. The following code listing shows how *WMRButtonDown* looks with the new function call added:

```
procedure ScribbleWindow.WMRButtonDown(var Msg: TMessage);
begin
  InvalidateRect(HWindow, nil, True);
  UpdateWindow(HWindow);
end;
```

Testing the fix

Run Color Scribble and exit it, then exit TDW. When you are back in Turbo Pascal, enter the changes to *WMRButtonDown*, then recompile the program and run it. Now when you draw on the window and then press the right mouse button to erase it, the window does get erased. You've found all the bugs, and the program now works perfectly.

A P P E N D I X

A

Error and information messages

TDW displays error messages and dialog boxes at the current cursor location. This chapter describes the dialog boxes and error and information messages TDW generates.

We tell you how to respond to both dialog box and error messages. All the dialog box messages and error messages (including the startup fatal error messages) are listed in alphabetical order, with a description provided for each one.

Dialog box messages

TDW displays a dialog box when you must supply additional information to complete a command. The title of the dialog box describes the information that's needed. The contents may show a history list (previous responses) that you have given.

You can respond to a dialog box in one of two ways:

- Enter a response and accept it by pressing *Enter.*
- Press *Esc* to cancel the dialog box and return to the menu command that preceded the dialog box.

Some dialog boxes only present a choice between two items (like Yes/No). You can use *Tab* to select the choice you want and then press *Enter,* or press *Y* or *N* directly. Cancel the command by pressing *Esc.*

For a more complete discussion of the keystroke commands to use when a dialog box is active, refer to Chapter 2.

Here's an alphabetical list of all the messages generated by dialog boxes:

Already recording, do you want to abort?
You are already recording a keystroke macro. You can't start recording another keystroke macro until you finish the current one. Press *Y* to stop recording the macro; *N* to continue recording the macro.

Device error – Retry?
An error has occurred while writing to a character device, such as the printer. This could be caused by the printer being unplugged, offline, or out of paper. Correct the condition and then press *Y* to retry or *N* to cancel the operation.

Disk error on drive __ – Retry?
A hardware error has occurred while accessing the indicated drive. This may mean you don't have a floppy disk in the drive or, in the case of a hard disk, it may indicate an unreadable or unwriteable portion of the disk. You can press *Y* to see if a retry will help; otherwise, press *N* to cancel the operation.

Edit watch expression
Modify or replace the watch expression. The dialog box is initialized to the currently highlighted watch expression.

Enter address, count, byte value
Enter the address of the block of memory you want to set to a particular byte value, then a comma, then the number of bytes you want to set, then another comma followed by the value to fill the block with.

Enter address to position to
Enter the address you want to view in your program. You can enter a function name, a line number, an absolute address, or a memory pointer expression. See Chapter 9 for more on entering addresses.

Enter animate delay (10ths of sec)
Specify how fast you want the Animate command to proceed. The higher the number, the longer between successive steps during animation.

Enter code address to execute to

Enter the address in your program where you want execution to stop. See Chapter 9 for more information on entering addresses.

Enter command-line arguments

Enter the command-line arguments for the program you're debugging.

Enter comment to add to end of log

Enter an arbitrary line of text to add to the messages displayed by the Log window. You can enter any text you want; it will be placed in the log exactly as you type it.

Enter destination address for marked block

Enter the segment:offset or segment that you want to move the marked block to.

Enter expression for conditional breakpoint

Enter an expression that must be true (nonzero) in order for the breakpoint to be triggered. This expression will be evaluated each time the breakpoint is encountered as your program executes. Be careful about any side effects it may have.

Enter expression to watch

Enter a variable name or expression whose value you want to watch in the Watches window. If you want, you can enter an expression that does not refer to a memory location, such as $x * y + 4$). If the dialog box is initialized from a text pane, you can accept the entry by pressing *Enter*, or change it and enter something else entirely.

Enter inspect start index, range

Enter the index of the first item in the array you want to view, followed by the number of items you want to view. Separate the two scalars by a space or a comma (,).

Enter instruction to assemble

Enter an assembler instruction to replace the one at the current address in the Code pane. The file ASMDEBUG.TDW has a condensed listing of all assembler keywords and discusses assembly language in more detail.

Enter log file name

Enter the name of the file you want to write the log to. Until you issue a Close Log File command, all lines sent to the log will be written to the file, as well as displayed in the window. The default file name has the extension .LOG and is the same file name as the program you are debugging. You can accept this name by pressing *Enter*, or type a new name instead.

Enter memory address, count

Enter a memory address, followed by an optional comma and the number of items you want to clear. You can use a symbol name or a complete expression for the address.

Enter name of file to view

You can use DOS-style wildcards to get a list of file choices, or you can type a specific file name to load.

Enter new bytes

Enter a byte list that will replace the bytes at the position in the file marked by the cursor. See Chapter 9 for a complete description of byte lists.

Enter new coprocessor register value

Enter a new value for the currently highlighted numeric coprocessor register. You can enter a full expression to generate the new value. The expression will be converted to the correct floating-point format before being loaded into the register.

Enter new data bytes

Enter a byte list to replace the bytes at the position in the segment marked by the cursor. See Chapter 9 for a complete description of byte lists.

Enter new directory

Enter the new drive or directory name that you want to become the current drive and directory.

Enter new file offset

You are viewing a disk file as hexadecimal data bytes. Enter the offset from the start of the file where you want to view the data bytes. The file will be positioned at the line that contains the offset you specified.

Enter new line number

Enter the line number you want to see in the current module. If you enter a line number that is past the end of the file, you'll see the last line in the file. Line numbers start at 1 for the first line in the file. The current line number that the cursor is on is shown as the first line of the Module window.

Enter new selector

Enter the selector value that you want to become current. You can enter an actual sector hex value, or you can enter a segment register value, such as CS, DS, or ES.

Enter new value

Enter a new value for the currently highlighted CPU register. You can enter a full expression to form the new value.

Enter port number

Enter the I/O port number you want to read from; valid port numbers are from 0 to 65,535.

Enter port number, value to output

Enter the I/O port number you want to write to, and the value to write; separate the two expressions with a comma. Valid port numbers are from 0 to 65,535.

Enter program name to load

Enter the name of a program to debug. You can use DOS wildcards to get a list of file choices, or you can type a specific file name to load. If you do not supply an extension to the file name, .EXE will be appended.

Enter read file name

Enter a file name or a wildcard specification for the file you want to read into memory. If you supply a wildcard specification or accept the default *.*, a list of matching files will be displayed for you to select from.

Enter search bytes

Enter a byte list to search for starting at the position in memory marked by the cursor. See Chapter 9 for a complete description of byte lists.

Enter search instruction or bytes

Enter an instruction, as you would for the Assemble local menu command, or enter a byte list as you would for a Search command in a Data pane.

Enter search string

Enter a character string to search for. You can use a simple wildcard matching facility to specify an inexact search string; for example, use * to match zero or more of any characters, and ? to match any single character.

Enter source address, destination, count

Enter the address of the block you want to move, the number of bytes to move, and the address you want to move them to. Separate the three expressions with commas.

Enter source directory path

Enter a list of directories, separated by spaces or semicolons (;). These directories will be searched, in the order that they appear in this list, for your source files.

Enter symbol table name

Enter the name of a symbol table to load from disk. Usually these files have an extension of .TDS. You must explicitly supply the filename extension.

Enter value to fill marked block

Enter a byte value to be filled into the marked block.

Enter variable to inspect

Enter the name of a variable or expression whose contents you want to examine. If the dialog box is initialized from a text pane, you can accept the entry by pressing *Enter* or change it and enter something else.

Enter write file name

Enter the name of the file you want to write the block of memory to.

Overwrite ___?

You have specified a file name to write to that already exists. You can choose to overwrite the file, replacing its previous contents, or you can cancel the command and leave the previous file intact.

Overwrite existing macro on selected key

You have pressed a key to record a macro, and that key already has a macro assigned to it. If you want to overwrite the existing macro, press *Y*; otherwise, press *N* to cancel the command.

Pick a method name

You have specified a routine name that can refer to more than one method in an object. Pick the correct one from the list presented.

Pick a module

Select a module name to view in the Module window. You are presented with a list of all the modules in your program. If you want to view a file that is not a program module, use View | File.

Pick a name

Pick a name from the list of displayed symbols. You can start to type a name, and you will be positioned to the first symbol, starting with what you have typed so far.

Pick a source file

Select a source file from the list displayed; only the source files that make up the current module are shown.

Pick a window

Pick a window from the list of open window titles.

Pick macro to delete

Pick the key or key combination for the macro you want to delete. The key will be returned to its original pre-macro functionality.

Press key to assign macro to

Press the key that you want to assign the macro to. Then, press the keys to do the command sequence that you want to assign to the macro key. The command sequence will actually be performed as you type it. To end the macro recording sequence, press the key you assigned the macro to, or press *Alt-*.

Program already terminated, Reload?

You have attempted to run or step your program after it has already terminated. If you choose *Y*, your program will be reloaded. If you choose *N*, your program will not be reloaded, and your run or step command will not be executed.

Reload program so arguments take effect?

You have just changed the command-line arguments for the program you're debugging. If you type *Y*, your program will be reloaded and set back to the start. You usually want to do this after changing the arguments because programs written in many Borland languages only look at their arguments once—just as the program is loaded. Any subsequent changes to the program arguments won't be noticed until the program is restarted.

Error messages

TDW uses error messages to tell you about things you haven't quite expected. Sometimes the command you have issued cannot be processed. At other times the message warns that things didn't go exactly as you wanted.

You can easily tell an error message from a prompt if you turn on Error Message Beeps in TDINST.

Fatal errors

All fatal errors cause TDW to quit and return to Windows. Some fatal errors are the result of trying to start TDW from the command line. A few others occur if something fatal happens while you are using the debugger. In either case, after having solved the problem, your only remedy is to restart TDW.

Bad or missing configuration file
The configuration file is either corrupted or not a TDW configuration file.

Cannot find *filename*.DLL
This message is generated by TDW for one of two reasons:

- You are attempting to load a program into TDW that requires one or more DLLs and TDW can't locate one of the .DLL files. The DLLs with symbol tables required by your executable must be in the same directory as the executable file.
- You are attempting to load TDW and the program can't find TDWIN.DLL. Either you have an invalid file name or path in the DebuggerDLL entry in TDW.INI, or if you don't have a TDW.INI, TDW is unable to find TDWIN.DLL in any of the places Windows knows to look.

  Either edit the DebuggerDLL entry in TDW.INI to reflect the correct path and file name, or if there is no TDW.INI, move TDWIN.INI to the main Windows directory.

Display adapter not supported by *filename*
The video driver *filename* indicated in the VideoDLL entry in TDW.INI does not support your display adapter. Check the README file for the correct video DLL and enter its path in the VideoDLL entry in TDW.INI.

Error loading *filename*

TDW was unable to load the video driver *filename*. The video driver you're loading could be an invalid driver file or it could be corrupted. Make sure the VideoDLL entry in TDW.INI represents a valid video driver. Check the README file for a list of valid drivers.

Invalid switch: __

You supplied an invalid option switch on the command line. Chapter 4 discusses each command-line option in detail.

Not enough memory

TDW ran out of working memory while loading.

Not enough memory to load *filename*

TDW ran out of working memory while loading the video driver *filename*.

Old configuration file

You have attempted to start TDW with a configuration file for a previous version. You must create new configuration files for this version of TDW.

Unsupported video adapter

TDW can't determine which display adapter you're using. CGA, EGA, VGA, and Hercules monographics are supported without special video driver support. If your adapter is SVGA or 8514, use one of the video DLLs listed in the README file.

Video mode not supported by *filename*

The video mode Windows is using isn't supported by the video DLL indicated in the VideoDLL entry in the TDW.INI file. Either change the Windows video mode or check the README file for an appropriate video DLL and enter that name in TDW.INI instead.

Other error messages

')' expected

While evaluating an expression, a right parenthesis was found to be missing. This happens if a correctly formed expression starts with a left parenthesis and does not end with a matching right one. For example,

```
3 * (7 + 4
```

should have been

```
3 * (7 + 4)
```

':' expected

While evaluating a C expression, a question mark (?) separating the first two expressions of the ternary **?:** operator was encountered; however, no matching : (colon) to separate the second and third expressions was found. For example,

```
x < 0 ? 4 6
```

should have been

```
x < 0 ? 4 : 6
```

']' expected

While evaluating an expression, a left bracket ([) starting an array index expression was encountered without a matching right bracket (]) to end the index expression. For example,

```
table[4
```

should have been

```
table[4]
```

This error can also occur when entering an assembler instruction using the built-in assembler. In this case, a left bracket was encountered that introduced a base or index register memory access, and there was no corresponding right bracket. For example,

```
mov ax,4[si
```

should have been

```
mov ax,4[si]
```

Already logging to a file

You issued an Open Log File command after having already issued the same command without an intervening Close Log File command. If you want to log to a different file, first close the current log by issuing the Close Log File command.

Ambiguous symbol name

You have entered a symbol name in an expression that does not uniquely identify a method in a Pascal program, and you have chosen not to pick the correct symbol from a list. You must pick the proper symbol from the list presented before your expression can be evaluated.

Bad or missing configuration file name

You have specified a nonexistent file name with the **–c** command-line option.

Cannot access an inactive scope

You entered an expression or pointed to a variable in a Module window that is not in an active function. Variables in inactive functions do not have a defined value, so you can't use them in expressions or look at their values.

Cannot be changed

You tried to change a symbol that can't be changed. The only symbols that can be changed directly are scalars (Byte, Integer, Longint, and String), pointers, and strings. If you want to change a record or array, you must change individual elements one at a time.

Can't have more than one segment override

You attempted to assemble an instruction where both operands have a segment override. Only one operand can have a segment override. For example,

```
mov es:[bx],ds:1
```

should have been one of the following:

```
mov es:[bx],1
```

or

```
mov ax,[1]
mov es:[bx],ax
```

Can't set a breakpoint at this location

You tried to set a breakpoint in ROM, nonexistent memory, or in segment 0. The only way to view a program executing in ROM is to use the Run | Trace Into command to watch it one instruction at a time.

Can't set any more hardware breakpoints

You can't set another hardware breakpoint without first deleting one you have already set. Different hardware debuggers support different numbers and types of hardware breakpoints.

Can't set hardware condition on this breakpoint

You've attempted to set a hardware condition on a breakpoint that isn't a global breakpoint. Hardware conditions can only be set on global breakpoints.

Can't set that sort of hardware breakpoint

The hardware device driver that you have installed in your CONFIG.SYS file can't do a hardware breakpoint with the combination of cycle type, address match, and data match that you have specified.

Constructors and destructors cannot be called

This error message appears only if you are debugging a program that uses objects. You probably tried to evaluate an object method that's either a constructor or a destructor. This is not allowed.

Count value too large

In the Data pane of the CPU window, you've entered too large a block length to one of the local menu Block commands. The block length can't exceed FFFFFh.

Ctrl-Alt-SysRq interrupt. System crash possible. Continue?

You attempted either to exit TDW or to reload your application program while the program was suspended as a result of your having pressed *Ctrl-Alt-SysRq*. Because Windows kernel code was executing at the time you suspended the application, exiting TDW or reloading the application will have unpredictable results (most likely hanging the system and forcing a reboot).

If possible, set a breakpoint in your code that will cause your program to exit to TDW, and then run your program again. When your program encounters the breakpoint and exits to TDW, you can terminate TDW or reload your program.

Destination too far away

You attempted to assemble a conditional jump instruction where the target address is too far from the current address. The target for a conditional jump instruction must be within –128 and 127 bytes of the instruction itself.

Divide by zero

You entered an expression using the divide (/, **div**) or modulus operators (**mod, %**) that had on its right side an expression that evaluated to zero. Since the divide and modulus operators do not have defined values in this case, an error message is issued.

DLL already in list

In the View | Modules dialog box, you tried to add a DLL to the DLLs & Programs list, but the DLL was already in the list.

Error opening file ___

TDW couldn't open the file that you want to look at in the File window. The file might not exist or might be in another directory.

Error opening log file___

The file name you supplied for the Open Log File local menu command can't be opened. Either there is not enough room to create the file, or the disk, directory path, or file name you specified is invalid. Either make room for the file by deleting some files from your disk, or supply a correct disk, path, and file name.

Error reading block into memory

The block you specified could not be read from the file into memory. You probably specified a byte count that exceeded the number of bytes in the file.

Error saving configuration

TDW could not write your configuration to disk. Make sure that there is some free space on your disk.

Error writing block to disk

The block that you specified could not be written to the file that you specified. You probably specified a count that exceeded the amount of free file space available on the disk.

Error writing log file ___

An error occurred while writing to the log file collecting the output from the log window. Your disk is probably full.

Error writing to file

TDW could not write your changes back to the file. The file might be marked as read-only, or a hard error may have occurred while writing to disk.

Exception HH

A CPU exception has occurred. The most common exceptions are 06 (Invalid Opcode) and 13 (0Dh—General Protection Fault, usually indicating a memory error, possibly caused by a bad pointer). The exeception numbers correspond to the CPU exception vector table.

Expression too complex

The expression you supplied is too complicated; you must supply an expression that has fewer operators and operands. You can have up to 45 operators and operands in an expression. Examples of operands are constants and variable names.

Examples of operators are plus (**+**), assignment (**:=**), and set membership (**in**).

Expression with side effects not permitted

You have entered an expression that modifies a memory location when it gets evaluated. You can't enter this type of expression whenever TDW might need to repeatedly evaluate an expression, such as when it is in an Inspector window or Watches window.

Extra input after expression

You entered an expression that was valid, but there was more text after the valid expression. This sometimes indicates that you omitted an operator in your expression. For example,

```
3 * 4 + 5 2
```

should have been

```
3 * 4 + 5 / 2
```

Another example,

```
add ax,4 5
```

should have been

```
add ax,45
```

You could also have entered a number in the wrong syntax for the language you are using, for example, 0xF000 instead of 0F000h when you are in assembler mode.

Fatal EMS error

Your EMS manager has reported a fatal error to Turbo Debugger.

Hardware device driver stuck

A hardware breakpoint is continuously triggering. Perhaps a hardware breakpoint has been set on a local variable (once a function returns, the local variables memory address becomes extremely volatile).

Help file ___ not found

You asked for help, but the disk file that contains the help screens could not be found. Make sure that the help file is in the same directory as the debugger program.

Illegal procedure or function call

You have attempted to evaluate a function at a time when you can't do so. This can happen in one of the following circumstances:

- You are attempting to call a function that is in an overlay.
- You are attempting to call an Object Pascal method that has been removed by the Turbo Pascal smart linker.

Immediate operand out of range

You entered an instruction that had a byte-sized operand combined with an immediate operand that is too large to fit in a byte. For example,

```
add BYTE PTR[bx],300
```

should have been

```
add WORD PTR[bx],300
```

Initialization not complete

You have attempted to access a variable in your program before the data segment has been set up properly by the compiler's initialization code. You must let the compiler initialization code execute to the start of your source code before you can access most program variables.

Invalid argument list

The expression you entered contains a procedure or function call that does not have a correctly formed argument list. An argument list starts with a left parenthesis, has zero or more comma-separated expressions for arguments, and ends with a right parenthesis.

Note that TDW requires empty parentheses to call a parameterless Pascal function or procedure. For example,

```
myfunc(1,2 3)
```

should have been

```
myfunc(1,2,3)
```

or

```
myfunc()
```

Invalid character constant

The expression you entered contains a badly formed character constant. A character constant consists of a single quote character (') followed by a single character, ending with another single quote character. For example,

```
'A = 'a'
```

should have been

```
'A' = 'a'
```

Invalid format string

You have entered a format control string after an expression, but it is not a valid format control string. See Chapter 9 for a description of format strings.

Invalid function parameter(s)

You have attempted to call a routine in an expression, but you have not supplied the proper parameters to the call.

Invalid instruction

You entered an instruction to assemble that had a valid instruction mnemonic, but the operand you supplied is not allowed. This usually happens if you attempt to assemble a **POP** CS instruction.

Invalid instruction mnemonic

When entering an instruction to be assembled, you failed to supply an instruction mnemonic. An instruction consists of an instruction mnemonic followed by optional arguments. For example,

```
AX,123
```

should have been

```
MOV ax,123
```

Invalid number entered

In a File window or a Module window, you typed an invalid number to go to (using the Goto command). Numbers must be greater than zero and in decimal format.

Invalid operand(s)

The instruction you're trying to assemble has one or more operands that aren't allowed. For example, a **MOV** instruction cannot have two operands that reference memory, and some instructions only work on word-sized operands. For example,

```
POP al
```

should have been

```
POP ax
```

Invalid operator/data combination

You've entered an expression where an operator has been given an operand that can't have the selected operation performed on it. For example, you attempt to multiply a constant by the address of a function in your program.

Invalid pass count entered

You have entered a breakpoint pass count that is not between 1 and 65,535. You can't set a pass count of 0. While your code is running, a pass count of 1 means that the breakpoint is eligible to be triggered the first time it is encountered.

Invalid register

You entered an invalid floating-point register as part of an instruction being assembled. A floating-point register consists of the letters *ST*, optionally followed by a number between 0 and 7 within parentheses; for example, ST or ST(4).

Invalid register combination in address expression

When entering an instruction to assemble, you supplied an operand that did not contain one of the permitted combinations of base and index registers. An address expression can contain a base register, an index register, or one of each. The base registers are BX and BP, and the index registers are SI and DI. Here are the valid address register combinations:

```
BX   BX+SI
BP   BP+SI
DI   BX+DI
SI   BP+DI
```

Invalid register in address expression

You entered an instruction to assemble that tried to use an invalid register as part of a memory address expression between brackets ([]). You can only use the BX, BP, SI, and DI registers in address expressions.

Invalid symbol in operand

When entering an instruction to assemble, you started an operand with a character that can never be used to start an operand: for example, the colon (:).

Invalid typecast

You entered an expression that contained an incorrectly formed typecast. A correct Pascal typecast starts with a known data type, then a left parenthesis, then an expression, then ends with a right parenthesis. For example,

```
Longint(p)
```

or

```
Word(p^)
```

Invalid value entered

When prompted to enter a memory address, you supplied a floating-point value instead of an integer value.

Keyword not a symbol (assembler only)

The expression you entered contains a keyword where a variable name was expected. You can only use keywords as part of typecast operations, with the exception of the **sizeof** special operator. For example,

```
floatval = char charval
```

should have been

```
floatval = char (charval)
```

Left side not a record, structure, or union

You entered an expression that used the Pascal record field qualifier (.). This symbol, however, was not preceded by a record name, nor was it preceded by a pointer to a record.

No coprocessor or emulator installed

You tried to create a Numeric Processor window using the View | Numeric Processor command, but there is no numeric processor chip installed on your system, and the program you're debugging either doesn't use the software emulator or the emulator has not been initialized.

No hardware debugging available

You have tried to set a breakpoint that requires hardware debugging support, but you don't have a hardware debugging device driver installed. You can also get this error if your hardware debugging device driver does not find the hardware it needs.

No help for this context

You pressed *F1* to get help, but TDW could not find a relevant help screen. Please report this to Borland technical support.

No modules have line number information

You have used the View | Module command, but TDW can't find any modules with enough debug information in them to let you look at any source modules. This message usually happens when you're debugging a program without a symbol table. See the "Program has no symbol table" error message entry on page 241 for more information on symbol tables.

No previous search expression

You attempted to perform a Next command from the local menu of a text pane, but you had not previously issued a Search command to specify what to search for. You can only use Next after issuing a Search command in a pane.

No program loaded

You attempted to issue a command that requires a program to be loaded. There are many commands that can only be issued when a program is loaded. For example, none of the commands in the Run menu can be performed without having a program loaded. Use the File | Open command to load a program before issuing these commands.

No type information for this symbol

You entered an expression that contains a program variable name without debug information attached to it. This can happen when the variable is in a module compiled without the correct debug information being generated. You can supply type information by preceding the variable name with a typecast expression to indicate its data type.

Not a function name

You entered an expression that contains a call to a routine, but the name preceding the left parenthesis introducing the call is not the name of a routine. Any time a parenthesis immediately follows a name, the expression parser presumes that you intend it to be a call to a routine.

Not a record, structure, or union member

You entered an expression that used the Pascal record field qualifier (.). This symbol, however, was not preceded by a record name, nor was it preceded by a pointer to a record.

Not enough memory for selected operation

You issued a command that needed to create a window, but there is not enough memory left for the new window. You must first remove or reduce the size of some of your windows before you can reissue the command.

Not enough memory to load program

Your program's symbol table has been successfully loaded into memory, but there is not enough memory left to load your program.

Not enough memory to load symbol table

There is not enough room to load your program's symbol table into memory. The symbol table contains the information that TDW uses when showing you your source code and program variables. If you have any resident utilities consuming memory, you might want to remove them and then restart TDW. You can also try making the symbol table smaller by having the compiler only generate debug information for those modules you are interested in debugging.

When this message is issued, you must free enough memory to load both your program and its symbol table. If you're debugging a TSR program that's already loaded, then you must start Turbo Debugger using the **–sm** command-line option to reserve memory for the program's symbol table.

Only one operand size allowed

You entered an instruction to assemble that had more than one size indicator. Once you have set the size of an operand, you can't change it. For example,

```
mov WORD PTR BYTE PTR[bx],1
```

should have been

```
mov BYTE PTR[bx],1
```

Operand must be memory location

You entered an expression that contained a subexpression that should have referenced a memory location but did not. An example of something that must reference memory is the assignment operator.

Operand size unknown

You entered an instruction to assemble, but did not specify the size of the operand. Some instructions that can act on bytes or words require you to specify which size to use if it cannot be deduced from the operands. For example,

```
add [bx],1
```

should have been

```
add BYTE PTR[bx],1
```

Path not found

You entered a drive and directory combination that does not exist. Check that you have specified the correct drive and that the directory path is spelled correctly.

Path or file not found

You specified a nonexistent or invalid file name or path when prompted for a file name to load. If you do not know the exact name of the file you want to load, you can pick the file name from a list by pressing *Enter* when the dialog box first appears. The names in the list that end with a backslash (\) are directories, letting you move up and down the directory tree through the lists.

Program has invalid symbol table

The symbol table attached to the end of your program has become corrupted. Re-create an .EXE file and reload it.

Program has no objects or classes

You've attempted to open a View | Hierarchy window on a program that isn't object-oriented.

Program has no symbol table

See page 59 for information on adding a symbol table to a program.

The program you want to debug has been successfully loaded, but it doesn't contain any debug symbol information. You'll still be able to step through the program using a CPU window to examine raw data, but you won't be able to refer to any code or data by name.

Program linked with wrong linker version

See page 59 for information on adding debug information to a program.

You are attempting to debug a program with out-of-date debug information. Relink your program using the latest version of the linker or recompile it with the latest version of the compiler.

Program not found

The program name you specified does not exist. Either supply the correct name or pick the program name from the file list.

Register cannot be used with this operator

You have entered an instruction to assemble that attempts to use a base or index register as a negative displacement. You can only use base and index registers as positive offsets. For example,

```
INC WORD PTR[12-BX]
```

should have been

```
INC WORD PTR[12+BX]
```

Register or displacement expected

You have entered an instruction to assemble that has a badly formed expression between brackets ([ ]). You can only put register names or constant displacement values between the brackets that form a base-indexed operand.

Run out of space for keystroke macros

The macro you are recording has run out of space. You can record up to 256 keystrokes for all macros.

Search expression not found

The text or bytes that you specified could not be found. The search starts at the current location in the file, as indicated by the cursor, and proceeds forward. If you want to search the entire file, press *Ctrl-PgUp* before issuing the search command.

Source file ___ not found

TDW can't find the source file for the module you want to examine. Before issuing this message, it has looked in several places:

- where the compiler found it
- in the directories specified by the **–sd** command-line option and the Options | Path for Source command
- in the current directory
- in the directory where TDW found the program you're debugging

You should add the directory that contains the source file to the directory search list by choosing Options | Path for Source.

Symbol not found

You entered an expression that contains an invalid variable name. You might have mistyped the variable name, or it might be in some procedure or function other than the active one or out of scope in a different module.

Symbol table file not found

The symbol table file that you have specified does not exist. You can specify either a .TDS or .EXE file for the symbol file.

Syntax error

You entered an expression in the wrong format. This is a general error message when a more specific message is not applicable.

Too many files match wildcard mask

You specified a wildcard file mask that included more than 100 files. Only the first 100 file names will be displayed.

Unexpected end of line

While evaluating an expression, the end of your expression was encountered before a valid expression was recognized.

For example,

```
99 - 22 *
```

should have been

```
99 - 22 * 4
```

And this example,

```
SUB AX,
```

should have been

```
SUB AX,4
```

Unknown character

You have entered an expression that contains a character that can never be used in an expression, such as a reverse single quote (‘).

Unknown record, structure, or union name

You have entered an expression that contains a typecast with an unknown record or enum name. (Note that assembler structures have their own name space different from variables.)

Unknown symbol

You entered an expression that contained an invalid local variable name. Either the module name is invalid, or the local symbol name or line number is incorrect.

Unterminated string

You entered a string that did not end with a closing quote ('). If you want to enter a string that contains quote characters in Pascal, it must contain additional quote characters (').

Value must be between *nn* and *nn*

You have entered an invalid numeric value for an editor setting (such as the tab width) or printer setting (such as the number of lines per page). The error message will tell you the allowed range of numbers.

Value out of range

You have entered a value for a Pascal variable that is outside the range of allowed values.

Variable not available

Your program's code has been optimized, and the variable you're looking for can no longer be accessed.

Video mode not available

You have attempted to switch to 43/50-line mode, but your display adapter does not support this mode; you can only use 43/50-line mode on an EGA or VGA.

I N D E X