2.0

USER'S GUIDE

TURBO DEBUGGER®

void run() /* The main program lo
displaycell(curcol, currow, HIGH
curcell = cell[curcol][currow];
showcelltype();
input = getkey();
DisplayChar:
cmp [byte
je Exit
CharacterOutput
inc
jmp
di

BORLAND

Turbo Debugger®

Version 2.0

User's Guide

BORLAND INTERNATIONAL, INC. 1800 GREEN HILLS ROAD
P.O. BOX 660001, SCOTTS VALLEY, CA 95066-0001

PRINTED IN THE USA.
R1 10 9 8

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 is a state-of-the-art, source-level debugger designed for Borland Turbo language programmers and programmers using other compilers who want a more powerful debugging environment.

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 Turbo Debugger's features:

- uses the expanded memory specification (EMS) for debugging large programs
- full C, Pascal, and assembler expression evaluation
- reconfigurable screen layout
- assembler/CPU access when needed
- powerful breakpoint and logging facility
- keystroke recording (macros)
- back tracing
- remote system for debugging large programs
- support for 80386 and other vendors' debugging hardware
- OOP full support for object-oriented programming in Turbo Pascal 5.5
- OOP full support for C++ in Turbo C++
- TSR and device driver debugging

Hardware and software requirements

Turbo Debugger runs on the IBM PC family of computers, including the XT and AT, the PS/2 series, and all true IBM compatibles. DOS 2.0 or higher is required and at least 384K of RAM. It runs on any 80-column monitor, either color or monochrome. We recommend a hard disk. If you want to run Turbo Debugger on a two-floppy system, you must use high-density disks. You can also use 3.5-inch, 720K disks; INSTALL won't install Turbo Debugger on these, so you will have to copy the files over yourself.

☞ Turbo Debugger does not require an 8087 math coprocessor chip.

☞ To use Turbo Debugger with Borland products, you must have Turbo Pascal 5.0 or later, Turbo C 2.0, Turbo C++, or Turbo Assembler 1.0 or later. You must already have compiled your source code into an executable (.EXE file) with full debugging information turned on.

☞ When you run Turbo Debugger, you'll need *both* the .EXE file and the original source files. Turbo Debugger 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 file is in.

A note on terminology

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

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

Function Refers to both a C function and to what is known in Pascal as a subprogram (or routine), which encompasses *functions*, *procedures*, and object *methods*. In C, a function can return a value (like a Pascal function) or not (like a Pascal procedure). (When a C function doesn't return a value, it's called a *void function*.) In the interest of brevity, we often use *function* in a generic way to stand for both C functions and Pascal functions and procedures—except, of course, in the language-specific areas of the manual.

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 procedures and 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 Turbo Debugger files into your system. It also gives you advice on which chapter to go to next, depending on your level of expertise.

Chapter 2: Debugging and Turbo Debugger explains the Turbo Debugger 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 either a Pascal or C program—that demonstrates many of the powerful capabilities of Turbo Debugger.

Chapter 4: Starting Turbo Debugger shows how to run the debugger from the DOS prompt, 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 Turbo Debugger 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 and modifying files describes how to examine and change program source files, as well as how to examine and modify arbitrary disk files, either as text or binary data.

Chapter 9: Expressions describes the syntax of C, 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: C++ and object-oriented Pascal debugging explains the debugger's special features that let you examine objects in Turbo Pascal 5.5 programs and classes in Turbo C++ programs.

Chapter 11: Assembler-level debugging explains how to view and change memory as raw hex data, how to use the built-in assembler and disassembler, and how to examine or modify the CPU registers and flags.

Chapter 12: The 80x87 coprocessor chip and emulator discusses how to examine and modify the contents of the floating-point hardware or emulator.

Chapter 13: Command reference is a complete listing of all main menu commands and all local menu commands for each window type.

Chapter 14: How to debug a program is an introduction to strategies for effective debugging of your programs.

Chapter 15: Virtual debugging on the 80386 processor describes how you can take advantage of the extended memory and power of an 80386 computer by letting the program you're debugging use the full address space below 640K, as if no debugger were loaded.

Chapter 16: Protected-mode debugging with TD286 tells you how to use TD286 to run Turbo Debugger in protected mode, freeing up memory for debugging large programs.

Chapter 17: Debugging TSRs and device drivers explains how to debug terminate and stay resident programs and programs that become resident at startup time with Turbo Debugger, and how to load a symbol table manually.

Appendix A: Summary of command-line options is a summary of all the command-line options that are completely described in Chapter 4.

Appendix B: Technical notes is for experienced programmers. It describes implementation details of Turbo Debugger that explain how it interacts with both your program and with DOS.

Appendix C: Inline assembler keywords lists all instruction mnemonics and other special words used for entering inline 8086/80286/80386 and 8087/80287/80837 instructions.

Appendix D: Customizing Turbo Debugger explains how to use the installation program (TDINST) to customize screen colors and change default options.

Appendix E: Remote debugging explains how to use the TDREMOTE utility so that you can run Turbo Debugger on one system and the program you are debugging on another.

Appendix F: Dialog boxes and error messages lists all the prompts and error messages that can occur, with suggestions on how to respond to them.

Appendix G: Using Turbo Debugger with different languages provides several tips when you're debugging programs written in C, assembler, or Pascal.

Glossary is an alphabetical list of commonly used terms in this manual, with short definitions.

How to contact Borland

The best way to contact Borland is to log on to Borland's Forum on CompuServe: Type `GO BOR` from the main CompuServe menu and choose "Borland Programming Forum B (Turbo Prolog, Turbo Assembler, Turbo Debugger, & Turbo C)" from the Borland main menu. Leave your questions or comments there for the support staff to process.

If you prefer, write a letter with your comments and send it to

Borland International
Technical Support Department – Turbo Debugger
1800 Green Hills Road
P.O. Box 660001
Scotts Valley, CA 95066-0001, USA

408-438-5300

You can also telephone our Technical Support department. Please have the following information handy before you call:

1. Product name and serial number on your original distribution disk. Please have your serial number ready, or we won't be able to process your call.
2. Product version number. The version number for Turbo Debugger is displayed when you first load the program and before you press any keys. If you are in Turbo Debugger, choose **A**bout from the ≡ (System) menu.

3. Computer brand, model, and the brands and model numbers of any additional hardware.
4. Operating system and version number. (The version number can be determined by typing `VER` at the MS-DOS prompt.)
5. Contents of your AUTOEXEC.BAT file.
6. Contents of your CONFIG.SYS file.

Recommended reading

Many leading publishers support Borland products with a wide range of excellent books, serving everyone from beginning programmers to advanced users. Here are a few titles that offer additional information on Turbo Debugger and Tools:

Ackerman, Charles. *Turbo Debugger and Tools: A Self-Teaching Guide,* John Wiley and Sons (New York: 1990).

Swan, Tom. *Mastering Turbo Assembler,* Howard W. Sams and Co. (Carmel, IN: 1989).

Swan, Tom. *Mastering Turbo Debugger and Tools,* Howard W. Sams and Co. (Carmel, IN: 1990).

Syck, Gary. *The Waite Group's Turbo Assembler Bible,* Howard W. Sams and Co. (Carmel, IN: 1990).

C H A P T E R

1

Getting started

Your Turbo Debugger package consists of a set of distribution disks and the *Turbo Debugger User's Guide* (this manual). The distribution disks contain all the programs, files, and utilities needed to debug programs written in Turbo C, Turbo Assembler, Turbo Pascal, and any program written with a Microsoft compiler. In the README and the HELPME!.DOC files, the Turbo Debugger package also contains documentation on subjects not covered in this manual.

The *Turbo Debugger User's Guide* provides a subject-by-subject introduction of Turbo Debugger's capabilities and a complete command reference.

Before you get started using Turbo Debugger, you should make a complete working copy of the distribution disks, then store the original disks in a safe place. Use the original distribution disks as your backup *only*, and run Turbo Debugger off of the copy you've just made—the distribution disks are your only backup in case anything happens to your working files.

If you are not familiar with Borland's no-nonsense license statement, now's the time to read the agreement. Mail your filled-in product registration card, so you'll be notified about updates and new products as they become available.

The distribution disks

When you install Turbo Debugger on your system, files from the distribution disks are copied to your working floppies or to your hard disk. Just run INSTALL.EXE, the easy-to-use installation program on your distribution disks. The distribution disks are formatted for double-sided, double-density disk drives and can be read by IBM PCs and close compatibles.

For a list of the files on your distribution disks, see the README file on the Installation disk.

The README file

☞ It is very important that you take the time to look at the README file on the Installation disk before you do anything else with Turbo Debugger. This file contains last-minute information that may not be in the manual. It also lists every file on the distribution disks, with a brief description of each.

To access the README file, insert the Installation disk in drive A, switch to drive A by typing `A:` and pressing *Enter*, then type `README` and press *Enter* again. Once you are in README, use the ↑ and ↓ keys to scroll through the file. Press *Esc* to exit.

The HELPME!.DOC file

Your Installation disk also contains a file called HELPME!.DOC, which contains answers to problems that users commonly run into. Consult it if you find yourself having difficulties. Among other things, the HELPME!.DOC file deals with:

- Screen output for graphics and text-based programs
- Executing other programs while you are still using the debugger

- Breaking out of a program
- The syntactic and parsing differences between Turbo Debugger and the Turbo languages
- Debugging multi-language programs with Turbo Debugger
- Tandy 1000A, IBM PC Convertible, or NEC MultiSpeed, and other computers that use the NMI (nonmaskable interrupt)

Turbo Debugger utilities

Your Turbo Debugger package comes with several utility programs. Detailed information on these utilities is available on your distribution disks. See the README file for how to access this disk-based documentation.

Here is a brief description of each of the Turbo Debugger utilities:

- The CodeView to Turbo Debugger utility, TDCONVRT.EXE, lets you debug C and assembler programs developed with Microsoft compilers.
- The remote file transfer utility, TDRF.EXE, works in conjunction with remote debugging and lets you issue basic file-maintenance commands to a remote system.
- The symbol table stripping utility, TDSTRIP.EXE, lets you strip the debugging information (the "symbol table") from your programs without relinking.
- TDPACK.EXE lets you pack the debugging information.
- TDMAP.EXE appends debugging information to a .MAP file.
- Finally, TDUMP.EXE is a generic object module and .EXE file disassembler program.
- Additionally, we give you a small TSR program, TDNMI.COM that resets the breakout-switch latch if you are using a Periscope I board.

For a list of all the command-line options available for TDCONVRT.EXE, TDRF.EXE, TDSTRIP.EXE, TDPACK.EXE, TDMAP.EXE, or TDUMP.EXE, just type the program name and press *Enter*. For example, to see the command-line options for TDMAP.EXE, you would enter

```
TDMAP
```

Installing Turbo Debugger

The Installation disks contain a program called INSTALL.EXE that will assist you with the installation of Turbo Debugger 2.0.

To start the installation, change your current drive to the one that has the INSTALL program on it and enter `INSTALL`. You are given instructions in a box at the bottom of the screen for each prompt.

INSTALL copies all Turbo Debugger files onto your hard disk and puts them into subdirectories. The default subdirectories are

Turbo Debugger directory:	C:\TD
Example subdirectory:	C:\TD

By default, all files from the distribution disks are placed in the Turbo Debugger directory. If you would rather separate the demo programs into their own subdirectory as well, edit the default example files path *before* selecting `START INSTALLATION`.

You should read the README file to get further information about Turbo Debugger after you install Turbo Debugger.

☞ For a list of all the command-line options available for INSTALL.EXE, enter the program name followed by `-h`:

```
INSTALL -h
```

Unzipping example files

The Turbo Debugger distribution disks contain a file with a .ZIP file name extension: TDEXAMPL.ZIP.

These files contain several other files that have been compressed and placed inside an archive. You can de-archive them yourself by using the UNZIP.EXE utility.

For example, entering

```
UNZIP TDEXAMPL
```

unpacks all the files stored in the TDEXAMPL.ZIP archive into the current directory.

INSTALL gives you a choice of copying the .ZIP files intact or de-archiving and copying all of the individual files onto your hard disk during the installation process.

LCD and B/W monitors

If you have difficulty reading the text displayed by the INSTALL utility, it accepts an optional **/B** command-line parameter that forces it to use black-and-white (BW80) mode:

```
A:INSTALL /B
```

Specifying the **/B** parameter may be necessary if you are using an LCD screen or a system that has a color graphics adapter and a monochrome or composite monitor.

Hardware debugging

If you're using an 80386 system, you can install the TDH386.SYS device driver supplied with Turbo Debugger. This device driver will vastly speed up breakpoints that watch for changed memory areas and I/O port accesses.

Copy this file to the directory where you keep your device drivers and put a line in your CONFIG.SYS file that loads the driver, such as

```
DEVICE = \SYS\TDH386.SYS
```

The next time you boot up your system, Turbo Debugger will be able to find and use this device driver.

See the disk-based documentation on the hardware debugger interface for complete information on this device driver interface.

☞ If you have a hardware debugging board (such as Atron, Periscope, Purart Trapper, and so on), you may be able to use the board with Turbo Debugger. Check with the vendor of your board for its compatibility with Turbo Debugger.

Where to now?

Now that you've loaded all the files, you can start learning about Turbo Debugger. Since this *User's Guide* is written for two types of users, different chapters of the manual may appeal to you. The following roadmap will guide you.

Programmers learning a Turbo language

If you are just starting to learn one of the languages in the Turbo family, you will want to be able to create small programs using it before you learn about the debugger. What better way to learn how to use the debugger than to have a real live problem of your own to debug! After you have 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 Turbo Debugger. 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.

Programmers already using a Turbo language

If you are an experienced Turbo family programmer, you can learn about the exciting new features of the Turbo Debugger environment by reading Chapter 2, "Debugging and Turbo Debugger." If it suits your style, you can then work through the tutorial or, if you prefer, move straight on to Chapter 4, "Starting Turbo Debugger." For a complete rundown of all commands, turn to Chapter 13, "Command reference."

C H A P T E R

2

Debugging and Turbo Debugger

The simple truth is that no one's perfect; we all make mistakes. Whether it's with simple things like walking or complicated things like programming, we all stumble sometimes.

If you're a programmer, stumbling is a way of life. You hardly ever write an error-free program the first time out the gate. That's nothing to be ashamed of. Stumbling also implies picking yourself up off the floor and trying again, and again, and maybe again. In programming parlance, that's debugging.

What is debugging?

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 an error
2. Finding where the error is
3. Finding the cause of the error
4. Fixing the error

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. Let's face it, you simply can't keep the entire program in your head at one time (unless it's a very small program indeed). 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 have only 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 down.

Fixing it

The final step is fixing the error. Armed with your knowledge of the program 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 14 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. The Borland language products have built-in syntax checkers that inform you of these errors and let 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 Turbo Debugger comes in.

What Turbo Debugger can do for you

Adding a full-feature debugger to the compiler itself would make it too big.

With the standalone Turbo Debugger, you have access to a much more powerful debugger than exists in your language compiler.

You must use a conversion utility that we supply before you debug a program written in a Microsoft language.

You can use Turbo Debugger with any program written in C, Pascal, or assembly language, either the Borland Turbo languages or those from other manufacturers if the compiler generates CodeView information. You can also debug any program created with another manufacturer's language product, but you'll be restricted to debugging on the assembly level—unless CodeView information is present. Then you must use the TDCONVRT utility described in the documentation on Turbo Debugger utilities on your distribution disks.

Turbo Debugger 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 slowing down 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 Turbo Debugger, you can perform *tracing, stepping, viewing, inspecting, changing,* and *watching.*

Tracing	You can execute your program one line at a time.
Back tracing	You can step backward through your executed code, reversing the execution as you go.
Stepping	You can execute your program one line at a time but step over any procedure or function calls. If you're sure your procedures and functions are

error-free, stepping over them speeds up debugging.

Viewing You can have Turbo Debugger open a special window to show you 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 You can have Turbo Debugger delve deeper into the workings of your program and show you the contents of complicated data structures like arrays.

Changing You can replace the current value of a variable, either globally or locally, with a value you specify.

Watching You can isolate program variables and keep 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, Turbo Debugger helps you find where it is and what's at fault.

OOP

Turbo Debugger 2.0 has even been enhanced to let you debug C++ and object-oriented Pascal programs. It is smart about objects and classes, and it correctly handles late binding of virtual methods or member functions so that it always executes and displays the correct code.

What Turbo Debugger won't do

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

- Turbo Debugger does not have a built-in editor to change your source code. Most programmers have their favorite editor and are comfortable with it. You can, however, easily transfer control to your text editor by choosing the local Edit command from a File window (more on local commands in a minute). Turbo Debugger uses the editor you specified with the TDINST installation program. Better still, if you have Turbo C++, you can use the new Transfer feature to run Turbo Debugger from inside the Turbo language's integrated environment.
- Turbo Debugger cannot recompile your program for you. You need the original program compiler (like Turbo Pascal or Turbo C) to do that.
- Turbo Debugger does not take the place of thinking. When you're debugging a program, your greatest asset is simple thought. Turbo Debugger is a powerful tool, but if you use it mindlessly, it's unlikely to save you time or effort.

How Turbo Debugger does it

Here's the really good news: Turbo Debugger gives you all this power and sophistication, and at the same time it's easy—dare we say intuitive—to use.

Turbo Debugger accomplishes this artful blend of power and ease by offering an exciting environment. The next section examines the advantages of Turbo Debugger's revolutionary environment.

The Turbo Debugger advantage

Once you start using Turbo Debugger, we think you'll be totally addicted to it. Turbo Debugger has been especially designed to be as easy and convenient as possible. To this end, Turbo Debugger offers you these powerful 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, Turbo Debugger 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.
- Convenient, complete window management.
- Mouse support.
- Access to several types of online help.
- Session recording and reverse execution.

The rest of this chapter discusses these six features of the Turbo Debugger environment.

Menus and dialog boxes

As with many Borland products, Turbo Debugger has a convenient global menu system accessible from a menu bar running along the top of the screen. This menu system is always available, no matter which of the debugger windows is *active* (that is, has a cursor in it).

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 a dialog box icon (...).

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 (*Spacebar, F, V, R, B, D, O, W, H*).
- Press *Alt* plus the first letter of any menu bar command (*Spacebar, F, V, R, B, D, O, W, H*). For example, wherever you are in the system, *Alt-F* takes you to the **F**ile 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 **View** 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* at any menu level (but *not* in a dialog box) to leave the menu system and return to the active window.
- Click the active window with the mouse to leave the menu system and return to the active window.

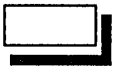
Some menu commands have a shortcut *hot key* that you press to execute them. The hot key appears in the menu to the right of these commands.

Figure 13.1 in Chapter 13 shows the complete pull-down menu tree for Turbo Debugger. Table 13.1 on page 192 lists all the hot keys. For a summary of all the commands available in Turbo Debugger, refer to Chapter 13.

Dialog boxes

Many of Turbo Debugger'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

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, Turbo Debugger 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 are multi-setting toggles that come in sets: You can choose only one radio button in a set at a time. When you do, a bullet appears between the parentheses: `(•)`.
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 the section "History lessons" for more on these).
THISFILE.EXE THATFILE.EXE TOTHERFL.EXE **List boxes**	A list box contains a list of items from which you can choose (for example, a list of possible files to open).

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

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 close 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're at

In addition to the convenient system of Borland pull-down menus, the Turbo Debugger 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, Turbo Debugger 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 Pascal program has a line like this:

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

As you'll discover when you work with Turbo Debugger, 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*, Turbo Debugger 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*, Turbo Debugger knows that you want to inspect one member and shows you only that member.

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

This sort of context-sensitivity makes Turbo Debugger 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). Turbo Debugger always does its best on delivering the goods for the particular item.

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

Local menus in Turbo Debugger 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 Turbo Debugger, however, you could never have both types of menus showing at the same time):

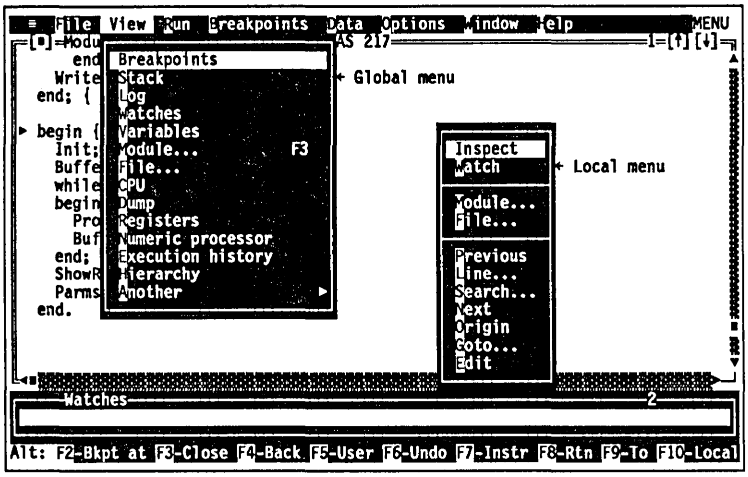


Figure 2.1
Global vs. local menus

Compare the following two lists:

Global menus

- Global menus are those that you access by pressing *F10* and using the arrow keys or typing the first letter of the menu name.
- The global menus are always available from the menu bar, visible at the top of the screen.
- Their contents never change.
- Some of the menu commands have hot key shortcuts that are available from any part of Turbo Debugger.

Local menus

- You call up a local menu by pressing *Alt-F10* or *Ctrl-F10*, or by clicking the right button on your mouse.
- The placement and contents of the menu depends on which window or pane you are in and where your cursor is.

- 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.
- Every command on a local menu has a hot key shortcut consisting of *Ctrl* plus the highlighted letter in the command.

  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 **G**oto command and the **S**earch 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 prevents you from trying to choose inappropriate commands and keeps the menus small and uncluttered.

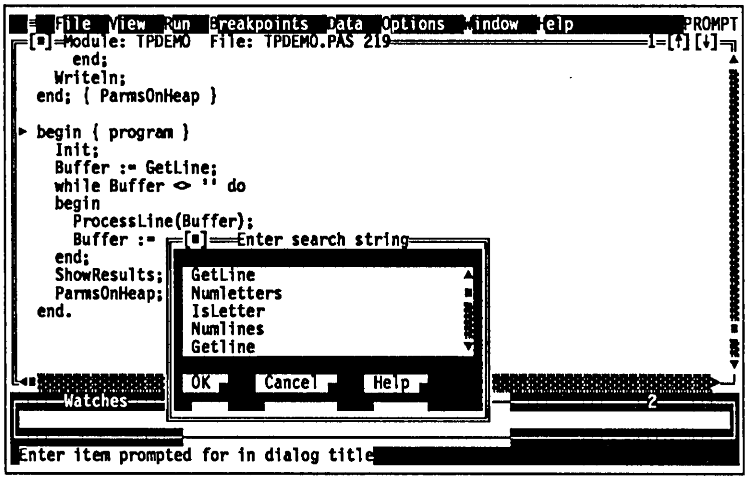
History lessons

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

Conforming to the philosophy that the user shouldn't have to type more than absolutely necessary, Turbo Debugger 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 search for a variable called *ReturnOnInvestment*. When you see the dialog box this time, you'll notice that *ReturnOnInvestment* 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 **S**earch command.

The search input box might look like this:



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

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

```
GetLine
NumLetters
IsLetter
NumLines
GetLine
```

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

Figure 2.2
A history list in an input box

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 five responses unless you tell it otherwise. (You can change its size using the TDINST program.)

Turbo Debugger 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.*

Warning!

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, Turbo Debugger 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

Turbo Debugger 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 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 that you define.*

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

```
Create         Alt=
Stop recording Alt-
Remove
Delete all
```

To create a macro, choose **O**ptions | **M**acros. At this point, you have a choice of four commands: **C**reate, **S**top Recording, **R**emove, and **D**elete All. Choose **C**reate; Turbo Debugger prompts you for a key to save the upcoming macro to. Press a little-used or easily remembered key or key combination (for example, *Shift-F1* for rerunning a program). Now go through all the steps and commands you want to save to that key.

To end the macro recording session, do one of these things:

- Choose **O**ptions | **M**acros | **S**top Recording.
- Press the newly defined macro key (*Shift-F1* in this example).
- Press *Alt* - (hold down *Alt* and press the hyphen or minus sign).

Window shopping

Lots of programs do windows these days, but Turbo Debugger does them better. Turbo Debugger 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 Turbo Debugger's windows come from the View menu, which lists fourteen 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

Here is a list of the thirteen types of windows that you can open from the View menu:

```
Breakpoints
Stack
Log
Watches
Variables
Module...          F3
File...
CPU
Dump
Registers
Numeric processor
Execution history
Hierarchy
Another            ►
```

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 in the section "Working with windows."

Module window

Displays the program code that you're debugging. You can move around inside the module 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.

Watches window

See Chapter 6 for more about the Watches window.

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

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 (in C programs, this is function **main**) and all subsequently called functions on top, in the order 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 Turbo Debugger. It tells you such things as why your program stopped, the results of breakpoints, and the contents of windows you saved in the log.

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

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.

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. You can search for specific text or byte sequences, as well as directly patching any part of the file on disk.

This is handy if you are debugging a program that uses disk files and you want to alter the program's behavior by changing the contents of one of its files. You can also correct a mistake in the contents of a file, or examine a file produced by a program to make sure the contents are correct.

CPU window

Chapter 11 discusses the CPU window and assembler-level debugging.

Displays the current state of the central processing unit (CPU). This window has five panes: one that contains disassembled machine instructions, 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.

Turbo Debugger sometimes opens a CPU window automatically, if your program stops on an instruction in the middle of a line of source code.

Dump window

See Chapter 11, 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, double words, 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. 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 11, 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 Chapter 12 for more information about using the Numeric Processor window.

Displays the current state of the math 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 math 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 assembly code and source lines for your program, up to the last line executed. The upper pane contains the assembly code that has been executed, so you can reverse back through it; the lower pane displays

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

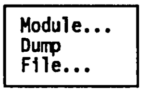
OOP

Lists and displays a hierarchy tree of all object or class types used by the current module. The window has two panes: one for the object/class type list, the other for the object/class hierarchy tree. (If you're debugging a C++ program with multiple inheritance, a third pane also opens, showing the parents of the highlighted class type.)

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

This window shows you the relationship of the objects or classes used by the current module. It also makes it possible for you to examine any object or class type, as well as its component data fields or members, and its methods or member functions, via its local menus.

Duplicate windows



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 Turbo Debugger 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 from DOS and not under Turbo Debugger.

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 (**int**, **float**, and so on), but also pointers, arrays, records, structures, and unions. Each type of data item is displayed in a way that closely mimics the way you are used to seeing it in your program's source code.

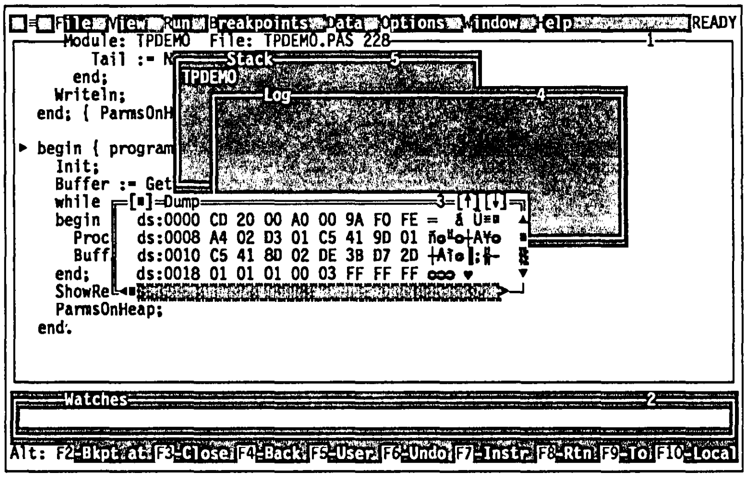
You create additional Inspector windows simply by choosing the **Inspect** command, whereas you can create additional Module, File, or CPU windows only by choosing **View | Another**.

The active window

Even though you can have many windows *open* in Turbo Debugger at the same time, only one window can be *active*. You can spot the active window by 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.



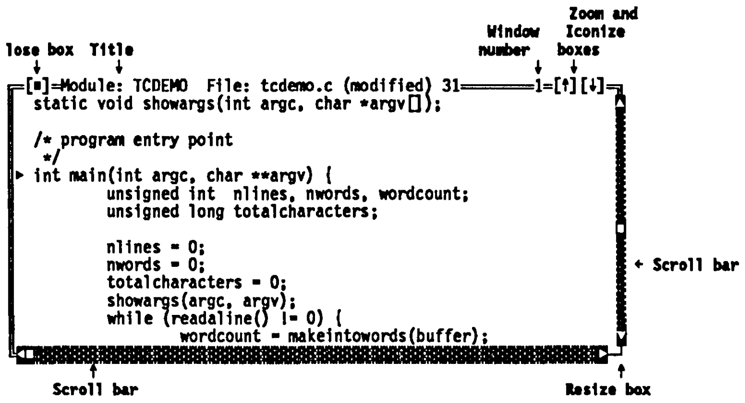
```
≡ File View Run Breakpoints Data Options Window Help                READY
  Module: TPDEMO  File: TPDEMO.PAS 228                              1
      Tail := N    Stack                     5
    end;           TPDEMO
  Writeln;                  Log                           4
end; { ParmsOnH
▶ begin { program
   Init;
   Buffer := Get
   while  [■]=Dump                           3=[↑][↓]
   begin    ds:0000 CD 20 00 A0 00 9A F0 FE =  å Ü≡■
     Proc   ds:0008 A4 02 D3 01 C5 41 9D 01 ño╙o┼A¥o
     Buff   ds:0010 C5 41 8D 02 DE 3B D7 2D ┼Aìo▐;╫-
   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
```

Figure 2.3 Can you spot the active window?

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: TCDEMO  File: tcdemo.c (modified) 31═══════1=[↑][↓]=
 static void showargs(int argc, char *argv[]);

 /* program entry point
  */
▶ int main(int argc, char **argv) {
         unsigned int  nlines, nwords, wordcount;
         unsigned long totalcharacters;

         nlines = 0;
         nwords = 0;
         totalcharacters = 0;
         showargs(argc, argv);
         while (readaline() != 0) {
                 wordcount = makeintowords(buffer);
```

- 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 this with your mouse to make the window larger or smaller.
- 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 iconize 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. Turbo Debugger 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 Windows menu.

Most of the window-management commands are in the **W**indows 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 **W**indow | **N**ext 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 **W**indow from the menu bar. The bottom half of the **W**indow 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 **W**indow 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 **W**indow | Ne**x**t Pane or pressing *Tab* or *Shift-Tab*.

You can also click the pane with the mouse.

The most pane-ful window in Turbo Debugger is the CPU window, which has five 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.) You can also use WordStar-like hot keys for moving around in the pane. Refer to Chapter 13 for a table of keystroke commands in panes.

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* to choose a command relevant to it from its local menu. 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. The exact function of the *Enter* key in these cases is described in the reference section starting on page 197.

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 **G**oto in a Module window, **S**earch in a File window, or **C**hange in a Registers window.

Moving and resizing windows

When you open a new window in Turbo Debugger, 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 in the upper right corner.

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**, or pressing *Alt-F3*, the hot key for this command.

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 Turbo Debugger 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) | **R**epaint Desktop.

Saving your window layout

Use the **O**ptions | **S**ave 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. The screen will then appear with your chosen layout each time you start Turbo Debugger from DOS, if the configuration has been saved to a file called TDCONFIG.TD. This is the only configuration file that is loaded automatically when Turbo Debugger is loaded. Other configurations *can* be loaded by using the **O**ptions | **R**estore Options command, if they have been saved to configuration files with a different name.

Getting help

As you've seen, Turbo Debugger 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 incredible control of windows. Even so, Turbo Debugger is a sophisticated program with lots of features and commands. To avoid potential confusion, Turbo Debugger 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 Turbo Debugger, look at the activity indicator for help. (Other activity indicator modes are `SIZE/MOVE`, `MOVE`, `ERROR`, `RECORDING`, `WAIT`, `RUNNING`, `MENU`, `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 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

Turbo Debugger, like other Borland products, gives context-sensitive onscreen help at the touch of a single key. Help is available anytime you're within a menu or window, as well as 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 in the status line. Some Help screens contain highlighted keywords that let you get additional help on that topic. Use *Tab* and *Shift-Tab* 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 **H**elp from the menu bar (*Alt-H*).

If you want to return to a previous Help screen, press *Alt-F1* or choose **P**revious from the Help menu. From within the Help system, use *PgUp* to scroll back through the last 20 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 **I**ndex from the Help menu. To get help on Help, choose **H**elp | **H**elp on Help. To exit from Help, press *Esc.*

You can get online help for reserved words via THELP.COM.

If you are using Turbo Pascal or Turbo C, and you want help on language-specific reserved words and functions such as you have in the integrated debuggers for these languages, you can get it via a RAM-resident utility called THELP.COM that comes with Turbo Pascal and Turbo C. To use THELP.COM,

1. Make sure that both THELP.COM and the help file for the language you are using (TURBO.HLP for Turbo Pascal, TCHELP.TCH for Turbo C) are copied into your Turbo Debugger directory or a directory on your path.
2. Type `THELP` and press *Enter.*
3. Go into Turbo Debugger.

4. To open a Help screen on any reserved word or function, position the cursor under the word you want help on, then press *5 on the numeric keypad.* (THELP won't work if you use the *5* on your keyboard.)
5. You can then use the help just as you would in the integrated debugger, paging through related screens, using *Alt-F1* to return to previous screens, and pressing *Enter* to bring up a screen on a selected keyword.
6. To exit the Help screen, press *Esc.*

☞ For more information on THELP, consult the THELP.DOC file for the Turbo language you are using.

The status line

Wherever you're in Turbo Debugger, 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 2.5
The normal status line

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

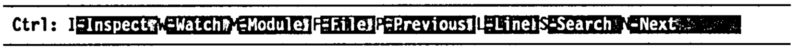
If you hold down the *Alt* key for a second or two, the commands performed by the *Alt* keys are displayed.

Figure 2.6
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* or *Ctrl-F10* to pop up the entire menu.

Figure 2.7
The status line with *Ctrl* pressed



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

If you have a mouse, all you have to do to execute an *Alt*- or *Ctrl*-key command is click the command in the status line.

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 | **R**egisters, 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.

C H A P T E R

3

A quick example

If you are itching to use Turbo Debugger 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 well-integrated, intuitive environment and context-sensitive help system let you learn as you go along.

This chapter leads you through all Turbo Debugger's basic features. After describing the demo programs—one in C and one in Pascal—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 programs

The demo programs (TCDEMO.C for C and TPDEMO.PAS for Pascal) introduce 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 programs themselves are not meant to be terribly useful: Some of their code and data structures exist solely to show you Turbo Debugger's capabilities.

Each demo program lets you type in some lines of text or the name of a data file, then counts the number of words and letters that you entered or that it reads from the file. At the end of the program, each displays some statistics about the text, including the average number of words per line and the frequency of each letter.

Make sure that your current directory contains the two files needed for each tutorial: TCDEMO.C and TCDEMO.EXE for the C example, TPDEMO.PAS and TPDEMO.EXE for the Pascal example.

Getting in To start the C program, enter

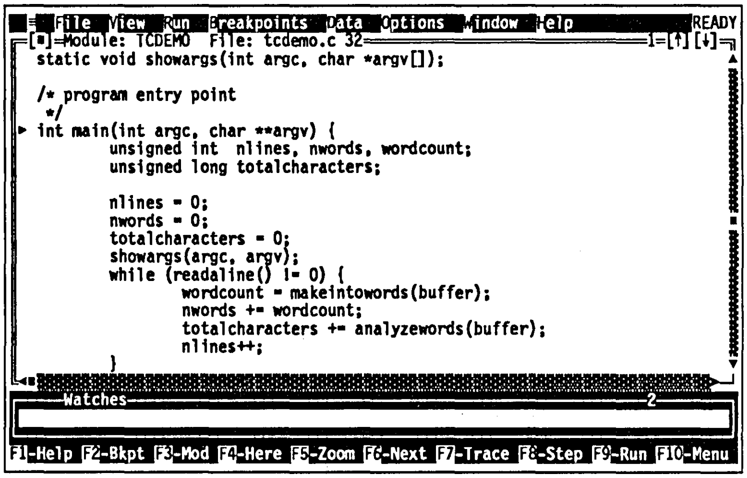
```
TD TCDEMO
```

To start the Pascal program, enter

```
TD TPDEMO
```

Turbo Debugger 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 TCDEMO



```
 ≡ File View Run Breakpoints Data Options Window Help                READY
[■]=Module: TCDEMO  File: tcdemo.c 32==============================1=[↑][↓]
  static void showargs(int argc, char *argv[]);

  /* program entry point
   */
► int main(int argc, char **argv) {
          unsigned int  nlines, nwords, wordcount;
          unsigned long totalcharacters;

          nlines = 0;
          nwords = 0;
          totalcharacters = 0;
          showargs(argc, argv);
          while (readaline() != 0) {
                  wordcount = makeintowords(buffer);
                  nwords += wordcount;
                  totalcharacters += analyzewords(buffer);
                  nlines++;
          }
 Watches                                                              2

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

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

Getting out To exit from the tutorial at any time and return to DOS, 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-F O* to do that. (*Alt-B D* deletes all breakpoints too, of course, but sometimes it's faster to reload with *Alt-F O*.)

Getting help

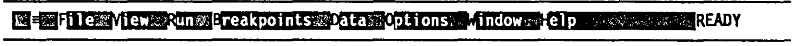
F1

Press *F1* whenever you need help about 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.

Using Turbo Debugger

The menus

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



≡ File 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 **File** command on the menu bar becomes highlighted. The bottom line of the screen also changes to indicate what sort of commands the **File** menu contains.

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

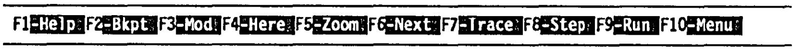
You can also open a menu by clicking an item in the menu bar with your mouse.

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.

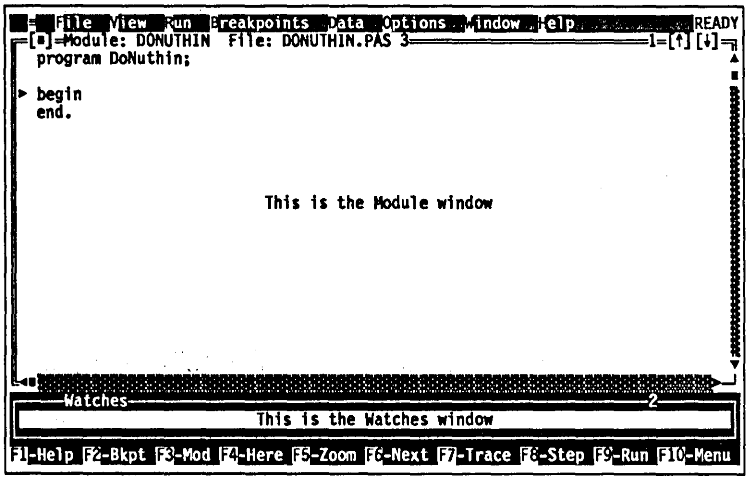
As soon as you enter the menu system, the status line changes again to show you what the currently highlighted menu option does. Press *F10* to go to the menu bar, and press → to highlight the **F**ile option. The status line now reads, "File oriented functions." Use ↓ to scroll through the options on the **F**ile 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 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*. This means they fill the entire screen without overlapping. New windows automatically overlap existing windows until you move them.

Figure 3.4
The Module and Watches windows, tiled



```
 ≡ File  View  Run  Breakpoints  Data  Options  Window  Help               READY
[■]=Module: DONUTHIN  File: DONUTHIN.PAS 3══════════════════════1=[↑][↓]
  program DoNuthin;

► begin
  end.


                    This is the Module window


 Watches ═══════════════════════════════════════════════════════2
                    This is the Watches window
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Notice that the Module window has a double-line border and a highlighted title. This means it is 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.

F6

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.

Turbo Debugger stores the last-closed window so you can recover it if you need to. If you accidentally close a window, choose **Window | Undo Close**. The Stack window reappears. You can also press *Alt-F6* to recover the last-closed window.

Alt F6

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 this too.)

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*. Depending on whether you've loaded the C or Pascal demo program, you should either continue with the next section (for the C sample) or move to the Pascal section on page 52.

Using the C demo program

The filled arrow (▶) in the left column of the Module window shows where Turbo Debugger 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.

F7

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.

As you can see from the **Run** menu, there are a number of ways to control the execution of your program. Let's say you want to execute the program until it reaches line 39.

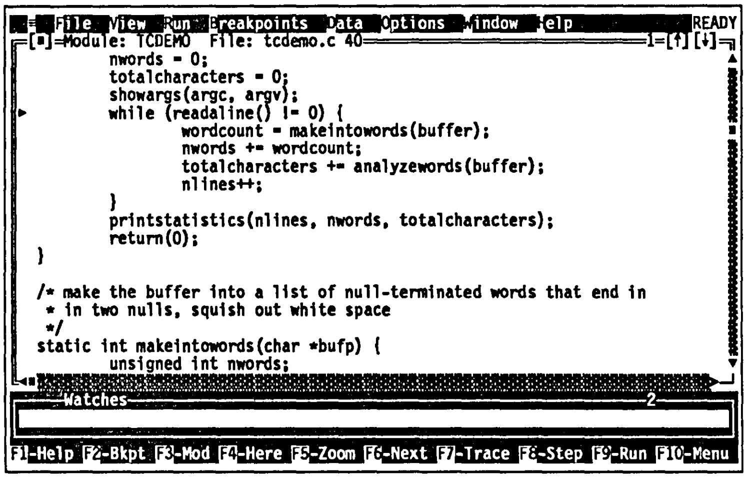
F4

First, position the cursor on line 39, then press *F4*. This runs the program up to (but not including) line 39. Now press *F7*, which executes one line of source code at a time; in this case, it executes line 39, a call to the function **showargs**. The cursor immediately jumps to line 151, where the definition of **showargs** is found. Continuing to press *F7* would step you through the function **showargs** and then return you to the line following the call—line 40. Instead, press *Alt-F8* to make the program stop when **showargs** returns. This too returns you to line 40. This command is very useful when you want to jump past the end of a function.

Alt F8

If you had pressed *F8* instead of *F7* on line 39, the cursor would have gone directly to line 40 instead of into the function. *F8* is similar to *F7* in that it executes functions, but it doesn't step through their source code.

Figure 3.5
Program stops on return from function showargs


≡ File View Run Breakpoints Data Options Window Help READY
Watches
2
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

```
[■]=Module: TCDEMO  File: tcdemo.c 40                    1=[↑][↓]
        nwords = 0;
        totalcharacters = 0;
        showargs(argc, argv);
►       while (readaline() != 0) {
                wordcount = makeintowords(buffer);
                nwords += wordcount;
                totalcharacters += analyzewords(buffer);
                nlines++;
        }
        printstatistics(nlines, nwords, totalcharacters);
        return(0);
}

/* make the buffer into a list of null-terminated words that end in
 * in two nulls, squish out white space
 */
static int makeintowords(char *bufp) {
        unsigned int nwords;
```

To execute the program until a specific place is reached, you can directly name the function or line number, without 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 `readaline` and press *Enter*. The program runs, then stops at the beginning of function **readaline** (line 142).

Alt F9

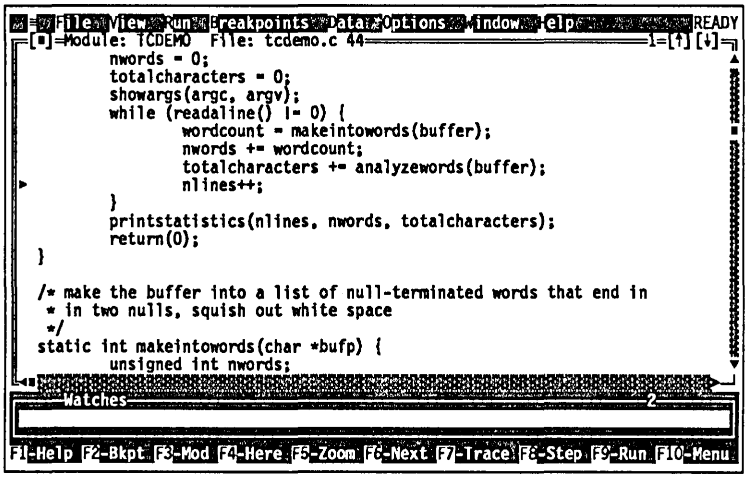
Setting breakpoints

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 44 and press *F2*. Turbo Debugger highlights the line, indicating there is a breakpoint set on it.

F2

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 44


File 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: TCDEMO  File: tcdemo.c 44                                  1=[↑][↓]
        nwords = 0;
        totalcharacters = 0;
        showargs(argc, argv);
        while (readaline() != 0) {
                wordcount = makeintowords(buffer);
                nwords += wordcount;
                totalcharacters += analyzewords(buffer);
►               nlines++;
        }
        printstatistics(nlines, nwords, totalcharacters);
        return(0);
}

/* make the buffer into a list of null-terminated words that end in
 * in two nulls, squish out white space
 */
static int makeintowords(char *bufp) {
        unsigned int nwords;
Watches                                                                     2
```

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 Turbo Debugger screen with the arrow on line 44, 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 *nwords*, move the cursor to the variable name on line 42 and choose Watch from the Module window local menu (bring it up with *Alt-F10* or choose the shortcut, *Ctrl-W*, from the status line).

Click *Ctrl-W* in the status line with your mouse.

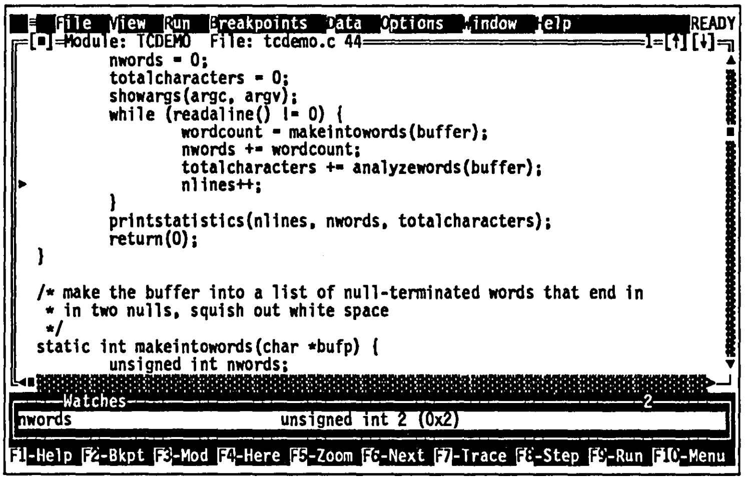


Figure 3.1
A C variable in the Watches window

```
≡ File View Run Breakpoints Data Options Window Help                  READY
[■]=Module: TCDEMO File: tcdemo.c 44                                  1=[↑][↓]
        nwords = 0;
        totalcharacters = 0;
        showargs(argc, argv);
        while (readaline() != 0) {
                wordcount = makeintowords(buffer);
                nwords += wordcount;
                totalcharacters += analyzewords(buffer);
►               nlines++;
        }
        printstatistics(nlines, nwords, totalcharacters);
        return(0);
}

/* make the buffer into a list of null-terminated words that end in
 * in two nulls, squish out white space
 */
static int makeintowords(char *bufp) {
        unsigned int nwords;
Watches                                                              2
nwords                      unsigned int 2 (0x2)
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

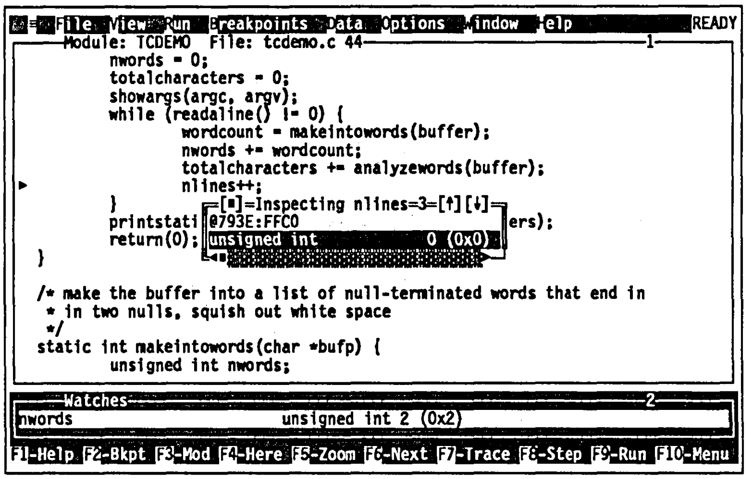
nwords now appears in the Watches window at the bottom of the screen, along with its type (**unsigned int**) and value. As you execute the program, Turbo Debugger updates this value to reflect the variable's current value.

Examining simple C 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 *nlines*. Move the cursor so it is under one of the letters in *nlines* and choose Inspect from the Module window local menu (press *Ctrl-I*). An Inspector window pops up.

Figure 3.8
An Inspector window


≡ File 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: TCDEMO  File: tcdemo.c 44                                  1
        nwords = 0;
        totalcharacters = 0;
        showargs(argc, argv);
        while (readaline() != 0) {
                wordcount = makeintowords(buffer);
                nwords += wordcount;
                totalcharacters += analyzewords(buffer);
▶               nlines++;
        }         [■]=Inspecting nlines=3=[↑][↓]
        printstati @793E:FFC0                    ers);
        return(0); unsigned int           0 (0x0)
}

/* make the buffer into a list of null-terminated words that end in
 * in two nulls, squish out white space
 */
static int makeintowords(char *bufp) {
        unsigned int nwords;

  Watches                                                            2
nwords                         unsigned int 2 (0x2)
```

The title tells you the variable name; the next line shows you its address in memory. The third line shows you what type of data is stored in *nlines* (it's a C **unsigned int**). To the right is 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.

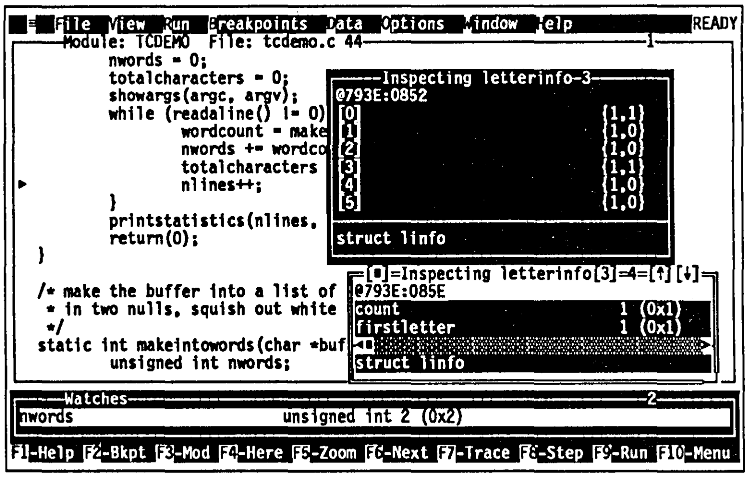
Let's review what you actually did here. By pressing *Ctrl*, you took a shortcut to the local menu commands in the Module window. Pressing *I* specified the Inspect command.

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 `letterinfo` and press *Enter*. An Inspector window appears, showing the values of the *letterinfo* array elements. The title of the Inspector window shows the name of the data you are inspecting. The first line under the title is the address in main memory of the first element of the array *letterinfo*. Use the arrow keys to scroll through the 26 elements that make up the *letterinfo* array. The next section shows you how to examine this compound data object.

Examining compound C data objects

A compound data object, such as an array or structure, contains multiple components. Move to the fourth element of the *letterinfo* array (the one indicated by `[3]`). Press *Alt-F10* to bring up the local menu for the Inspector window, then press *I* to choose Inspect. A new Inspector window appears, showing the contents of that element in the array. This Inspector window shows the contents of a structure of type *linfo*.

Figure 3.9
Inspecting a structure



```
 ≡ File  View  Run  Breakpoints  Data  Options  Window  Help              READY
   Module: TCDEMO  File: tcdemo.c 44──────────────────────────────────1─
         nwords = 0;
         totalcharacters = 0;      ────Inspecting letterinfo-3──────
         showargs(argc, argv);     @793E:0852
         while (readaline() != 0)  [0]                          {1,1}
                wordcount = make   [1]                          {1,0}
                nwords += wordco   [2]                          {1,0}
                totalcharacters    [3]                          {1,1}
 ▶              nlines++;          [4]                          {1,0}
         }                         [5]                          {1,0}
         printstatistics(nlines,
         return(0);                struct linfo
 }
                                    [■]=Inspecting letterinfo[3]=4=[↑][↓]
 /* make the buffer into a list of  @793E:085E
  * in two nulls, squish out white  count                   1 (0x1)
  */                                firstletter             1 (0x1)
 static int makeintowords(char *buf ◄■░░░░░░░░░░░░░░░░░░░░░░░░░░░░►
         unsigned int nwords;       struct linfo

  ───Watches──────────────────────────────────────────────────2─
 nwords                      unsigned int 2 (0x2)

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

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.

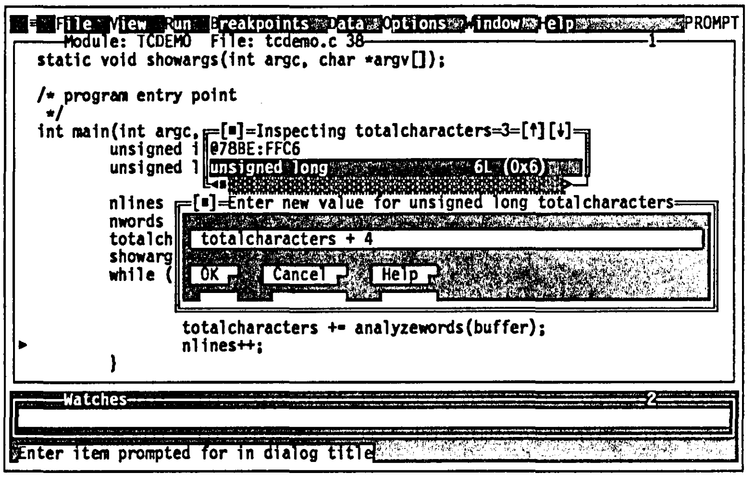
Alt F3

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 latest Inspector window would have been deleted.)

Changing C 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 38 in the source file. Place the cursor at the variable *totalcharacters* and press *Ctrl-I* to inspect its value. With the Inspector window open, press *Alt-F10* to bring up the Inspector's local menu, and choose the **C**hange option. (You could also have done this directly by pressing *Ctrl-C*.) A dialog box appears, asking for the new value.



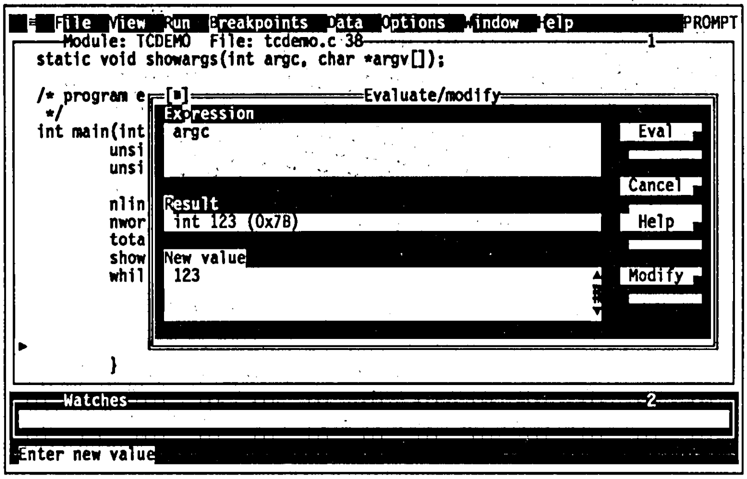
```
≡ File View Run Breakpoints Data Options Window Help                 PROMPT
┌─Module: TCDEMO  File: tcdemo.c 38──────────────────────────────1──┐
  static void showargs(int argc, char *argv[]);

  /* program entry point
   */
  int main(int argc, ┌[■]=Inspecting totalcharacters=3=[↑][↓]┐
          unsigned i │@78BE:FFC6
          unsigned l │unsigned long                 6L (0x6)
                     └◄■                                    ►┘
          nlines  ┌[■]=Enter new value for unsigned long totalcharacters┐
          nwords  │
          totalch │ totalcharacters + 4
          showarg │
          while ( │ OK    Cancel    Help
                  └
                totalcharacters += analyzewords(buffer);
 ►              nlines++;
          }
┌─Watches──────────────────────────────────────────────────────2──┐

Enter item prompted for in dialog title
```

Figure 3.10
The Change dialog box

At this point, you can enter any C expression that evaluates to a number. Type `totalcharacters + 4` and press *Enter*. The value in the Inspector window now shows the new value, `10L (0xA)`.

To change a data item that isn't displayed in the Module window, choose **D**ata | **E**valuate/Modify. A dialog box appears. Enter the name of the variable to change in the first input box: Type `argc` and press *Enter*. Then press *Tab* twice to move to the input box labeled New Value. Type `123` and press *Enter*. The result (second box) changes to `int 123 (0x7B)`.

Figure 3.11
The Evaluate/Modify dialog box



```
≡ File View Run Breakpoints Data Options Window Help            PROMPT
 Module: TCDEMO  File: tcdemo.c 38                                  1
 static void showargs(int argc, char *argv[]);

 /* program e [■]              Evaluate/modify
  */            Expression
 int main(int    argc                                        Eval
         unsi
         unsi                                                Cancel

         nlin   Result
         nwor    int 123 (0x7B)                              Help
         tota
         show   New value
         whil    123                                         Modify

▶
         }
 Watches                                                            2

Enter new value
```

That's a quick introduction to using the Turbo Debugger with a Turbo C program. Chapter 14 offers a more extensive debugging sample.

Using the Pascal sample program

The filled arrow (▶) in the left column of the Module window shows where Turbo Debugger 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.

F7

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 221, move the cursor to that line and then press *F4*. TPDEMO prompts you to enter a string. Type ABC, a space, DEF, and then press *Enter*. Now, with the cursor still on line 221, 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 224. Instead, press *Alt-F8* to make the program stop when *ProcessLine* returns. This command is very

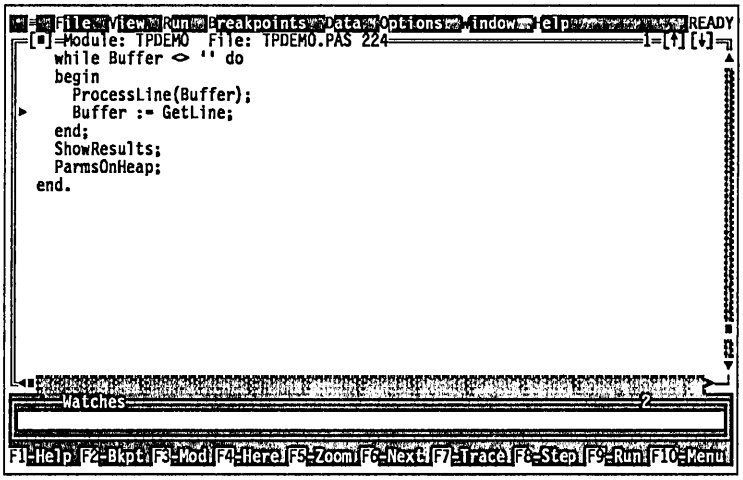
Alt F8

useful when you want to jump past the end of a function or procedure.

F8

If you had pressed *F8* instead of *F7* on line 221, the cursor would have gone directly to line 224 instead of into the function. *F8* is similar to *F7* in that it executes functions, but it doesn't step through their source code.

Figure 3.12
The program stops after returning from a procedure


≡ File View Run Breakpoints Data Options Window Help READY
[■]=Module: TPDEMO File: TPDEMO.PAS 224 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.
```

Alt F9

To execute the program until a specific place is reached, you can directly name the function or line number, without 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*.

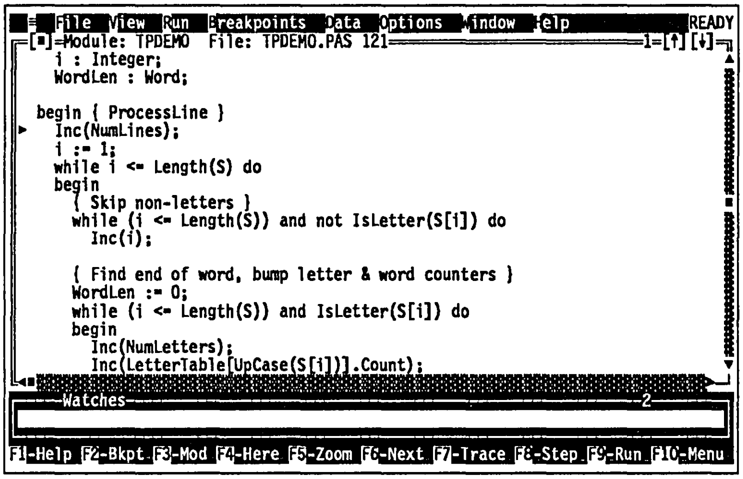
Setting breakpoints

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 121 and press *F2*. Turbo Debugger highlights the line, indicating there is a breakpoint set on it.

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

Figure 3.13
A breakpoint at line 121



```
≡ File View Run Breakpoints Data Options Window Help                READY
[■]=Module: TPDEMO  File: TPDEMO.PAS 121==========================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 Turbo Debugger screen with the arrow on line 121, 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.

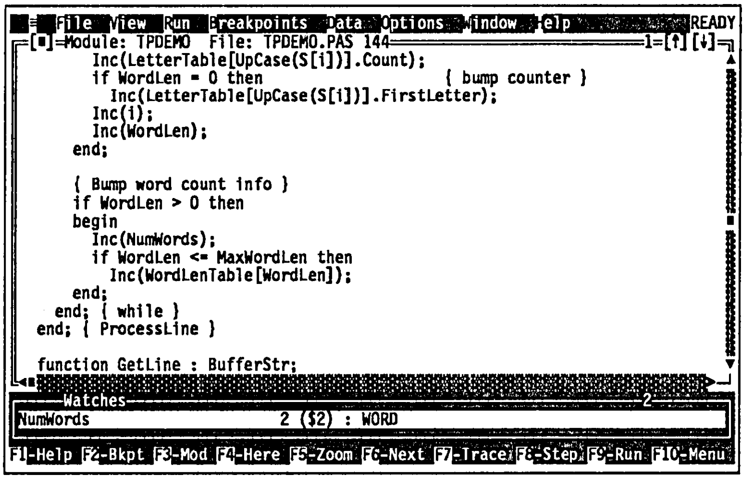
Using watches

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 144 and choose Watch from the Module window local menu (bring it up with *Alt-F10*, or choose the shortcut, *Ctrl-W*, from the status line).

You can also click *Ctrl-W* in the status line with your mouse.

Figure 3.1
A Pascal variable in the Watches window



```
≡ File  View  Run  Breakpoints  Data  Options  Window  Help                READY
╔═[■]═Module: TPDEMO  File: TPDEMO.PAS 144══════════════════════1═[↑][↓]═╗
          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;
─────Watches──────────────────────────────────────────────────────2─────
NumWords                      2 ($2) : WORD
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

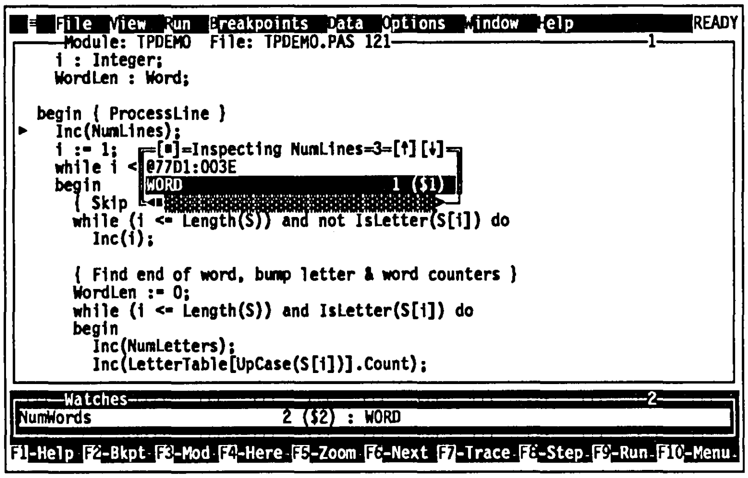
NumWords now appears in the Watches window at the bottom of the screen, along with its type (Word) and value. As you execute the program, Turbo Debugger updates this value to reflect the variable's current value.

Examining simple Pascal data objects

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

The **I**nspect commands (in various local menus and in the **D**ata 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 121 so it's under one of the letters in *NumLines* and press *Ctrl-I*. An Inspector window pops up.

Figure 3.15
An Inspector window



```
 ≡ File View Run Breakpoints Data Options Window Help                  READY
┌──Module: TPDEMO  File: TPDEMO.PAS 121─────────────────────────────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
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

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.

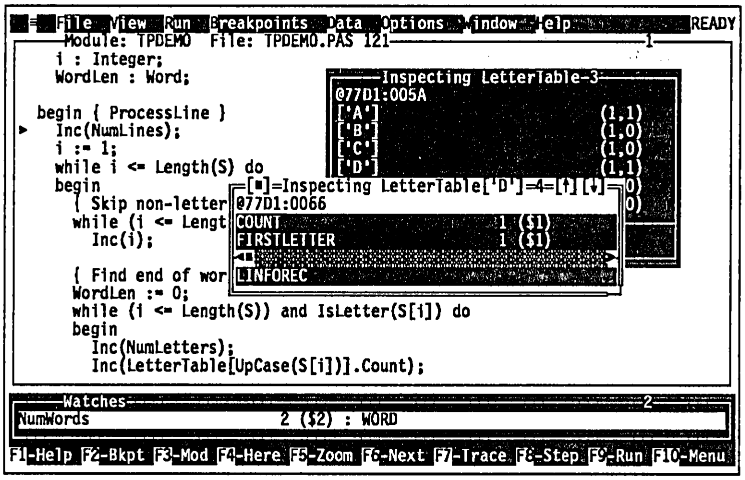
Let's review what you actually did here. By pressing *Ctrl*, you used a hot key for the local menu commands in the Module window. Pressing *I* specified the **I**nspect command.

To examine a data item that is not conveniently displayed in the Module window, choose **D**ata | **I**nspect. 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 and type of the data you are inspecting, exactly as the declaration for this data appears in the source file. 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 3.16
Inspecting a record



```
 ≡ File  View  Run  Breakpoints  Data  Options  Window  Help                READY
  Module: TPDEMO  File: TPDEMO.PAS 121                                      1
     i : Integer;
     WordLen : Word;                       Inspecting LetterTable-3
                                          @77D1:005A
   begin { ProcessLine }                  ['A']                    (1,1)
 ►   Inc(NumLines);                       ['B']                    (1,0)
     i := 1;                              ['C']                    (1,0)
     while i <= Length(S) do              ['D']                    (1,1)
     begin                     [■]=Inspecting LetterTable['D']=4=[↑][↓]0)
       { Skip non-letter @77D1:0066                                 0)
       while (i <= Lengt COUNT                       1 ($1)
         Inc(i);         FIRSTLETTER                 1 ($1)

       { Find end of wor LINFOREC
       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
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

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

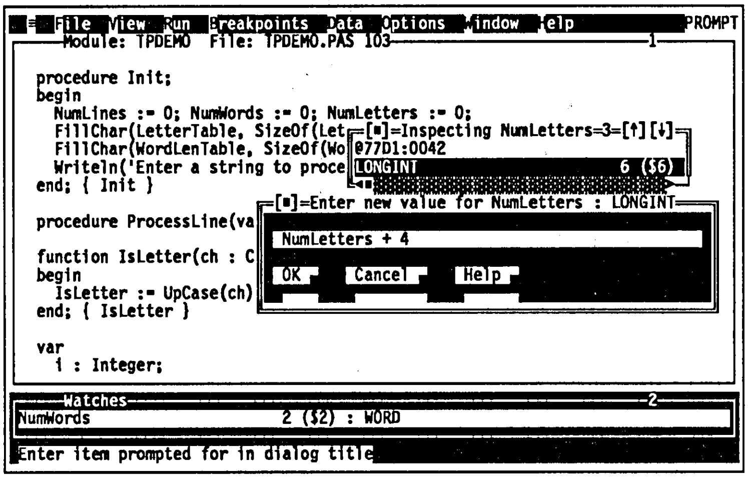
Press *Alt-F3* to remove both Inspector windows and return to the Module window. (*Alt-F3* is a convenient way of removing several Inspector windows at once. If you had pressed *Esc*, only the topmost Inspector window would have been deleted.)

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

Figure 3.17
The Change dialog box

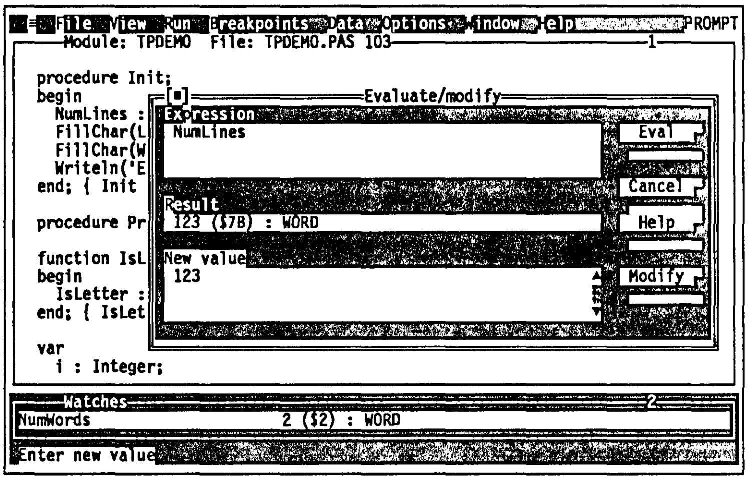


```
Module: TPDEMO  File: TPDEMO.PAS 103                                  1

procedure Init;
begin
  NumLines := 0; NumWords := 0; NumLetters := 0;
  FillChar(LetterTable, SizeOf(Let  [■]=Inspecting NumLetters=3=[↑][↓]
  FillChar(WordLenTable, SizeOf(Wo  @77D1:0042
  Writeln('Enter a string to proce  LONGINT                      6 ($6)
end; { Init }
                                  [■]=Enter new value for NumLetters : LONGINT
procedure ProcessLine(va
                                    NumLetters + 4
function IsLetter(ch : C
begin                               OK    Cancel    Help
  IsLetter := UpCase(ch)
end; { IsLetter }

var
  i : Integer;

Watches                                                                2
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`.

To change a data item that isn't displayed in the Module window, choose **D**ata | **E**valuate/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 3.18
The Evaluate/Modify dialog box



```
Module: TPDEMO  File: TPDEMO.PAS 103                                 1
procedure Init;
begin
  NumLines :
  FillChar(L
  FillChar(W
  Writeln('E
end; { Init

procedure Pr

function IsL
begin
  IsLetter :
end; { IsLet

var
  i : Integer;
```

```
NumLines
```

```
123 ($7B) : WORD
```

```
123
```

```
NumWords                          2 ($2) : WORD
Enter new value
```

That wraps up our quick introduction to using Turbo Debugger with a Turbo Pascal program. Chapter 14 offers a more extensive debugging sample.

C H A P T E R

4

Starting Turbo Debugger

This chapter tells you how to prepare programs for debugging. We show you how to start Turbo Debugger from the DOS command line, 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. You also learn how to run a DOS command processor from within a Turbo Debugger session and, finally, how to return to DOS when you are done.

Preparing programs for debugging

When you compile and link with one of Borland's Turbo languages, you can tell the compiler to generate full debugging information. If you have compiled your program's object modules without any debugging information, you must recompile all its modules to have full source debugging capabilities throughout your program. It is 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.

Preparing Turbo C programs

If you're using the Turbo C++ integrated environment (TC), open the Debugger dialog box (choose **Options | Debugger**) and set the Source Debugging radio button to Standalone before you compile your source modules. For Turbo C 2.0, set **Debug | Source Debugging** to Standalone.

If you're using the command-line compiler (TCC), specify the **–v** command-line option.

If you're using TLINK as a standalone linker, you must use the **/v** option to append debugging information at the end of the .EXE file.

You also should make sure optimizing is disabled. Either don't use the **–O** option or specify **–O–** to turn off the **–O** in your TURBOC.CFG file. This eliminates the few occasions when Turbo Debugger appears to skip over lines of source code when you're stepping through a program.

Preparing Turbo Pascal programs

First, make sure that you have version 5.0 or later of Turbo Pascal. Earlier versions do not have the ability to bundle debugging information into the .EXE file so that Turbo Debugger can use it.

If you're using the integrated environment (TURBO.EXE), go to the **Debug** menu and change the Standalone Debugging setting to *On*. Turn **Options | Compiler | Debug Information** *On* or use the {$D+} compiler directive. If you want to be able to reference local symbols (any declared within procedures and functions), you must either set **Options | Compiler | Local Symbols** to *On* or put this directive at the start of your program:

Just like this, with no spaces

```
{$L+}
```

You can then compile your program.

If you're using the command-line version (TPC.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, you can use **/$** command-line options to disable them.

Preparing Turbo Assembler programs

To debug a Turbo Assembler program, specify the **–zi** command-line option to get full debugging information.

To link your program with TLINK, use the **/v** option to append debugging information at the end of the .EXE file.

Preparing Microsoft programs

See the documentation on your distribution disks for information about how to use the utility program TDCONVRT.EXE, which converts CodeView executable programs to Turbo Debugger format.

Running Turbo Debugger

To debug a program with Turbo Debugger, simply type `TD` at the DOS prompt, followed by an optional set of command-line arguments and the name of the program, and press *Enter*. Turbo Debugger then loads your program, displaying its source code so you can step through your program statement by statement.

The generic command-line format is

```
TD [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 example command lines:

Command	**Action**
`td -sc prog1 a b`	Starts the debugger with **–sc** option and loads program *prog1* with two command-line arguments, *a* and *b*.
`td prog2 -x`	Starts the debugger with default options and loads program *prog2* with one argument, *–x*.

If you simply type `TD` *Enter*, Turbo Debugger loads and uses its default options.

☞ When you run a program in Turbo Debugger, you need to have *both* its .EXE file and the original source files available. Turbo Debugger searches for source files first in the directory the compiler found them in when it compiled, second in the directory specified in the **O**ptions | **P**ath for Source command, third in the current directory, and fourth in the directory the .EXE file is in.

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

☞ Remember, Turbo Debugger works only with programs in Turbo Pascal 5.0 or later, Turbo C 2.0 or later, or Turbo Assembler 1.0 or later.

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

Command-line options

Appendix A has an easy-to-use list of Turbo Debugger's command-line options.

All Turbo Debugger command-line options start with a hyphen (-) and are separated from the TD 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, **-vg-** turns off a complete graphics save. You can do this if an option has been permanently enabled in the configuration file. You can modify the configuration file by using the TDINST configuration program described in Appendix D.

The following sections describe all available 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.TD is loaded if it exists. Here's an example:

```
TD -cMYCONF.TD TCDEMO
```

This loads the configuration file MYCONF.TD and the source code for TCDEMO.

Display updating (-d)

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

–do	Runs Turbo Debugger on your secondary display. View your program's screen on the primary display, and run the debugger on the secondary one.
–dp	The default option for color displays. Shows the debugger on one display page and the program being debugged on another, minimizing the time it takes to swap between the two screens. You can use this option only on a display that has multiple display pages. You can't use this option if the program you are debugging uses multiple display pages itself. This is the default for display updating.
–ds	The default option for monochrome displays. 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 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 screenful of help that describes Turbo Debugger's command-line syntax and options.

Process ID switching (-i)

This option enables process ID switching. Don't use this option when you are debugging inside DOS or when DOS system calls are active. See Appendix B for more technical information on this feature. You needn't be concerned with this option to debug most programs.

Keystroke recording (-k)

This option enables keystroke recording in the Keystroke Recording pane of the Execution History window.

If you use this option, all keystrokes that you type during a debugging session will be recorded to a disk file. Then you can recover to a previous point in your debugging session by having Turbo Debugger reload your program and play back the recorded keystrokes. Turbo Debugger records both the keys you press while you're in Turbo Debugger and the keys you press while your program is running.

Assembler-mode startup (-l)

This option forces startup in assembler mode, showing the CPU window. Turbo Debugger 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.

Setting heap size (-m)

This option sets the working heap used by Turbo Debugger to *N*K, where the syntax is

```
-mN
```

and *N* is the number of kilobytes. A space must not exist between the **–m** option and the size of the heap. Here's an example:

```
TD -m10 TCDEMO.EXE
```

The default heap size is 18K; the low boundary is 7K. If you need memory, use this option to reduce the amount of heap Turbo Debugger uses. Turbo Debugger stores transient information, such as command history lists and breakpoints, in the heap.

☞ If you specify a heap size of 0 (zero) with the **–m** command-line option (**-m**0), Turbo Debugger uses the maximum that it's able to use, usually 18K.

Mouse support (-p)

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

Remote debugging (-r)

All **–r** options affect the remote debugging link.

–r Enables debugging on a remote system over the serial link. Uses the default serial port (COM1) and speed (115 Kbaud), unless you have changed them with TDINST.

–rp*N* Sets the remote link port to port *N*. *N* can be 1 or 2 to indicate COM1 or COM2, respectively.

–rs*N* Sets the remote link speed. *N* can be 1 for 9600 baud, 2 for 40 Kbaud, or 3 for 115 Kbaud.

Source code handling (-s)

All **–s** options affect the way Turbo Debugger handles source code and program identifiers.

This option does not affect Pascal, because it is not case-sensitive.

–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.

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

```
-sddirname
```

To set multiple directories, use the **–sd** option repeatedly—only one directory name can be specified with each **–sd** option. Directories are searched 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.

–sm*N* This option sets the symbol table reserved memory size. Follow it with the number of kilobytes you want to reserve, like this:

```
-smN
```

where *N* is the number of kilobytes. Use this option if you want to load a symbol table manually with the

File | **Symbol** Load command. You may have to experiment with the amount of memory to reserve.

Video hardware (-v)

All **-v** options affect how Turbo Debugger handles the video hardware.

-vg Saves complete graphics image on program screen. Requires an extra 8K of memory, but can debug programs that use certain graphics display modes. Try this option if your program's graphics screen becomes corrupted when running under Turbo Debugger.

-vn 43/50-line display is not allowed. Specifying this option saves some memory. Use this if you're running on an EGA or VGA and know you won't switch into 43- or 50-line mode once Turbo Debugger is running.

-vp Enables the EGA palette save.

Overlay pool size (-y)

The **-y** options are used to set the size of the overlay pool size, either in main memory or in EMS memory.

-y*N* This option sets the overlay pool size in main memory. The syntax is as follows, where *N* is the number of kilobytes you want to reserve:

```
-yN
```

Use TDINST to set a permanent overlay code pool size.

Normally, Turbo Debugger uses a 80K code pool size. The smallest pool size that you can set is 20K, and the largest is 200K.

Use this option if you do not have enough memory to load your program under Turbo Debugger, or if you are debugging small programs and want to improve Turbo Debugger's performance. The smaller the code pool size, the more often Turbo Debugger loads program overlays from disk, and the slower it responds. With a larger code pool, there is less memory available for the program you are debugging, but Turbo Debugger runs faster.

-ye*N* This option sets the overlay pool size in EMS memory. Use this option if you need to free up some EMS memory for the program you are debugging. The syntax is as follows, where *N* is the number of 16K EMS pages you want to reserve:

```
-ye*N*
```

For example, **–ye**4 sets the overlay pool to four pages. The default is twelve 16K EMS pages.

Use **–ye**0 to disable the EMS overlay pool.

Configuration files

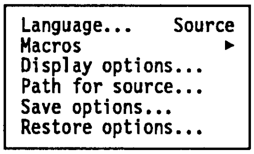
Turbo Debugger uses a configuration file to override built-in default values for command-line options. You can use TDINST to set the options that Turbo Debugger will default to if there is no configuration file. You can also use it to build configuration files.

Turbo Debugger looks for the configuration file TDCONFIG.TD first in the current directory, next in the TURBO directory set up with the TDINST installation program, and then in the directory that contains TD.EXE. If you are running on DOS 2.x, Turbo Debugger won't look for TDCONFIG.TD in the TD.EXE directory.

Appendix D describes how to use the installation program to create configuration files.

If Turbo Debugger finds a configuration file, the settings in that file override its built-in defaults. Any command-line options that you supply when you start Turbo Debugger from DOS override those default options and any values in TDCONFIG.TD.

The Options menu

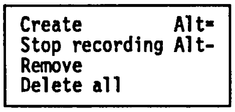


The Options menu lets you set or adjust a number of parameters that control the overall appearance and operation of Turbo Debugger. The following sections describe each menu command and refer you to other sections of the manual where you can find more details.

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 **M**acros 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: **C**reate, **S**top Recording, **R**emove, and **D**elete All.

Create

Starts recording keystrokes that you are assigning to a key (for example, *Alt-M*). To begin a recording session, choose **O**ptions | **M**acros | **C**reate. You are prompted for the key you want to assign the macro to. The message `RECORDING` is displayed in the upper right-hand corner of the screen while the recording session is in progress. Type the keystrokes you want to record. These keystrokes are acted upon by Turbo Debugger exactly as if you were not recording a macro.

Once you have finished recording keystrokes, issue the **O**ptions | **M**acros | **S**top Recording command or its hot key, *Alt-Hyphen*. You can also press the key you assigned the macro to (*Alt-M*) once more.

Alt = is the hot key for starting to record a macro.

Stop Recording

Stops recording keystrokes that are assigned to a key. Use this command after issuing the **O**ptions | **M**acros | **C**reate command to assign keystrokes to a key.

Alt-Hyphen is the hot key for ending a macro.

Remove

Removes a macro assigned to a single key. You are prompted to press the key of the macro you want to delete.

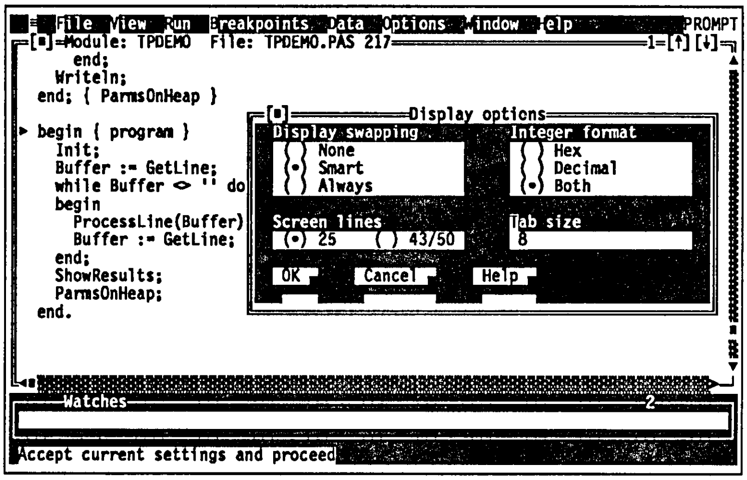
Delete All

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 Turbo Debugger display.

Figure 4.1
The Display Options dialog box


```
≡ File View Run Breakpoints Data Options Window Help                PROMPT
[■]=Module: TPDEMO  File: TPDEMO.PAS 217=========================1=[↑][↓]
      end;
    Writeln;
  end; { ParmsOnHeap }
                          [■]=========Display options=========
► begin { program }        Display swapping       Integer format
    Init;                   ( ) None               ( ) Hex
    Buffer := GetLine;      (•) Smart              ( ) Decimal
    while Buffer <> '' do   ( ) Always             (•) Both
    begin
      ProcessLine(Buffer)   Screen lines           Tab size
      Buffer := GetLine;    (•) 25    ( ) 43/50    8
    end;
    ShowResults;            OK     Cancel     Help
    ParmsOnHeap;
  end.

Watches                                                            2

Accept current settings and proceed
```

Display Swapping

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

None Don't swap between the two screens. Use this option if you're debugging a program that does not output to the User screen.

Smart Swap to the User screen only when display output may occur. Turbo Debugger swaps the screens any time that you step over a routine, or if you execute an instruction or source line that appears to read from or write to video memory. This is the default option.

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, since Turbo Debugger's screen is replaced for a short time with the User screen.

Integer Format

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

Decimal	Shows integers as ordinary decimal numbers.
Hex	Shows integers as hexadecimal numbers, displayed in a format appropriate to the current language.
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 Turbo Debugger'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

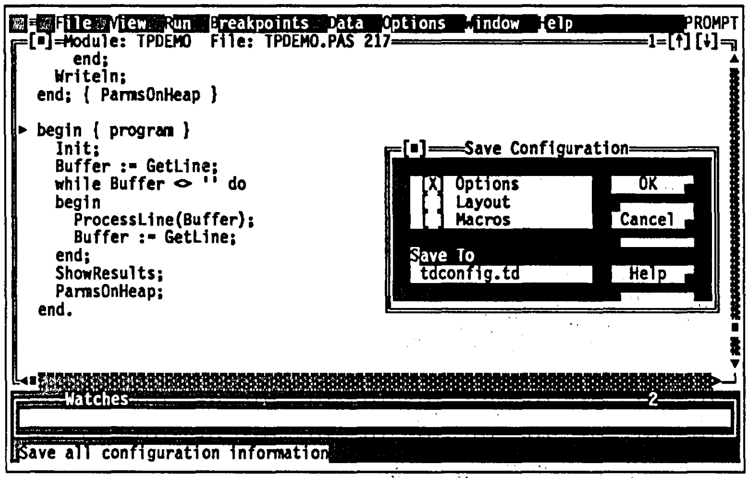
Sets the directories that Turbo Debugger searches for your source files. See the discussion of the Module window in Chapter 8 for more information.

Save Options command

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

- your macros
- the current window layout and pane formats
- all settings made in the **O**ptions menu

Figure 4.2
The Save Options dialog box



```
≡ File View Run Breakpoints Data Options Window Help                PROMPT
╔[■]═Module: TPDEMO  File: TPDEMO.PAS 217══════════════════════1═[↑][↓]╗
║     end;
║   Writeln;
║  end; { ParmsOnHeap }
║
► begin { program }
║   Init;                                 ╔[■]═══Save Configuration═════╗
║   Buffer := GetLine;                    ║                             ║
║   while Buffer <> '' do                 ║ [X] Options           OK    ║
║   begin                                 ║ [ ] Layout                  ║
║     ProcessLine(Buffer);                ║ [ ] Macros           Cancel ║
║     Buffer := GetLine;                  ║                             ║
║   end;                                  ║ Save To                     ║
║   ShowResults;                          ║  tdconfig.td          Help  ║
║   ParmsOnHeap;                          ║                             ║
║  end.                                   ╚═════════════════════════════╝
╚═══════════════════════════════════════════════════════════════════════╝
╔═Watches════════════════════════════════════════════════════════2═════╗
╚══════════════════════════════════════════════════════════════════════╝
Save all configuration information
```

Turbo Debugger 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 **O**ptions menu.
Layout	Saves only the windowing layout.
Macros	Saves only 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 by the **Sa**ve Options command or with TDINST.

Running DOS in Turbo Debugger

When debugging a program, you sometimes need to use another program or utility. Do this via **File | DOS** Shell.

When you start the DOS command processor, the program you are debugging is swapped to disk if necessary. This lets you perform DOS commands even while you are debugging a program that takes all the available memory. Of course, this means that there may be a few seconds of delay while your program is being swapped to and from the disk.

Warning! Do not load TSRs (terminate and stay resident programs) on top of Turbo Debugger while you are shelled to DOS.

When you have finished issuing commands to DOS, type `EXIT` and press *Enter* to return to your debugging session.

Returning to DOS

You can end your debugging session and return to DOS 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**.

All the memory initially allocated to the program being debugged is freed. If the program you are debugging allocates memory via the DOS block memory allocation routines, that memory is also freed.

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. Turbo Debugger 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 Turbo Debugger encounters a breakpoint.

This chapter shows you how to examine the state of your program whenever Turbo Debugger 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 DOS 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 **O**rigin and **G**et 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 **V**iew | **V**ariables. 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.

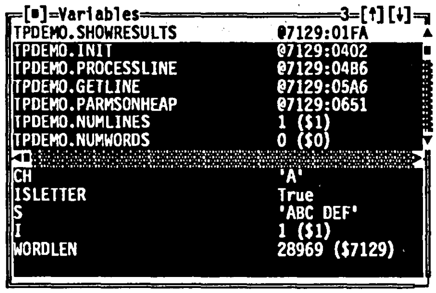


Figure 5.1
The Variables window

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

When you're debugging a Turbo Pascal program, the variables won't be arranged alphabetically.

You open a Variables window by choosing **View | Variables**. A 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 Turbo Debugger 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 functions 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 two commands: **Inspect** and **Change**.

Inspect

```
Inspect
Change
```

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 at the Change dialog box. Turbo Debugger 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.

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
```

The Static pane has these two local menu commands: **Inspect** and **Change**.

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

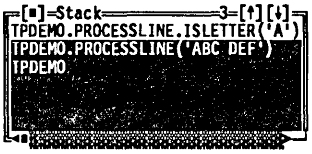
See Chapter 9 for more information on assignment and data type conversion.

Changes the value of the currently selected (highlighted) local symbol to the value you enter at the Change dialog box. Turbo Debugger performs any data type conversion necessary, 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 (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.

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 first function or procedure in the program (the main program in Pascal; in C programs, usually the function called **main**). For each procedure or function, you see the value of each parameter it was called with.



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

Figure 5.2
The Stack window

OOP

The Stack window likewise displays the names of object methods or class member functions, prefixed with the name of the object or class type that defines the method or member function:

```
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: **I**nspect and **L**ocals.

Inspect

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

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

Locals

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

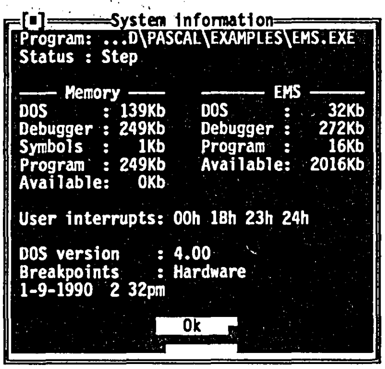
The Origin local menu command

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 and other information appears in a text box (see Figure 0) that disappears with your next keystroke:

- The name of the program you're debugging.
- A description of why your program stopped.
- The amount of memory used by DOS, Turbo Debugger, and your program.
- If you have EMS memory, its use appears to the right of main memory use.
- A list of interrupts intercepted by the program you're debugging.
- The DOS version you're running.
- Whether breakpoints are handled entirely in software or if they have hardware assistance.



```
[■]═══════System Information══════════
Program: ...D\PASCAL\EXAMPLES\EMS.EXE
Status : Step

─── Memory ───        ───── EMS ─────
DOS       : 139Kb     DOS       :   32Kb
Debugger  : 249Kb     Debugger  :  272Kb
Symbols   :   1Kb     Program   :   16Kb
Program   : 249Kb     Available: 2016Kb
Available:    0Kb

User interrupts: 00h 18h 23h 24h

DOS version    : 4.00
Breakpoints    : Hardware
1-9-1990  2 32pm

              Ok
```

Figure 5.3
The Get Info text box

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 Turbo Debugger 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.

Control Break
You interrupted execution of your program with *Ctrl-Break*.

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

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 occurred.

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

Loaded
You loaded Turbo Debugger and specified a program *and* the option that prevents the compiler startup code from executing. No instructions have been executed at this point, including those that set up your stack and segment registers. This means that if you try to examine certain data in your program, you may see incorrect values.

Step
You executed a single source line or machine instruction, skipping function calls, with *F8* (**R**un | **S**tep Over).

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

Exception __
You were using TD386, and a processor exception has occurred. This usually happens when your program attempts to execute an illegal instruction opcode. The Intel processor documentation describes each exception code in complete detail.

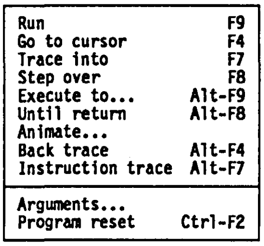
Hardware device driver stuck
You were using a hardware debugger and set a hardware breakpoint in a stack variable that is conflicting with Turbo Debugger. You must remove the hardware breakpoint before you proceed.

Divide by zero
Your program has executed a divide instruction where the dividend 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 **R**un menu has a number of options for executing different parts of your program. Since you use these options frequently, they are all 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 the debugger when one of the following events occurs:

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

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, Turbo Debugger traces into the routine. However, if the current window is a CPU window, only a single machine instruction is executed.

OOP

Turbo Debugger treats object methods and class member functions 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 call(s). If the current window is a Module window, this command usually executes a single source line. However, if the current window is a CPU window, only a single machine instruction is executed.

If you step over a single source line, Turbo Debugger treats any function or procedure call(s) in that line as part of the line. You don't end up at the start of one of the functions. 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, Turbo Debugger treats certain instructions as a single instruction, even when they cause multiple assembly instructions to be executed. Here is a complete list of the instructions Turbo Debugger 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 or a class member function 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 may never be reached if a breakpoint action is encountered first, or if you interrupt execution.

Until Return

Alt F8

Executes until the current function 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 function 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. You interrupt this command by pressing any key.

After you choose **Run** | **Animate**, you will be prompted 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

If you are tracing (*F7* or *Alt-F7*) through your program, reverses the order of execution. This 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 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 Instructions pane of the Execution History window.

Warning! Some restrictions apply. See the section, "The Instructions pane (page 86)."

Instruction Trace

Alt F7

Executes a single machine instruction. Use this 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. This is discussed more in Chapter 5.

Program Reset

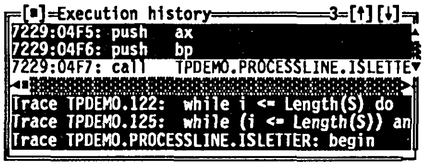
Reloads from disk the program you're debugging. Use this when you've executed past the place where you think there is a bug.

Ctrl F2

If you're in a Module or CPU window, the debugger won't return to the start of the program. Instead, you'll stay exactly where you were when you chose the **P**rogram Reset command. If you chose **P**rogram 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, or choose **R**un | **B**ack Trace to step back through previously executed code instead of choosing **P**rogram Reset.

The Execution History window

Turbo Debugger has a special feature called the *execution history* that keeps track of each instruction as it is executed (provided you are tracing into the code), and also, if you want, records the keystrokes you input to get to a given point in your program. You can examine these instructions, and also undo them to return to a point in the program where you think there might be a bug. If you don't have EMS memory, Turbo Debugger can record about 400 instructions. If you have EMS, it can record approximately 3000 instructions.



```
[■]=Execution history=============3=[↑][↓]
7229:04F5: push   ax
7229:04F6: push   bp
7229:04F7: call   TPDEMO.PROCESSLINE.ISLETTE
Trace TPDEMO.122:  while i <= Length(S) do
Trace TPDEMO.125:  while (i <= Length(S)) an
Trace TPDEMO.PROCESSLINE.ISLETTER: begin
```

Figure 5.4
The Execution History window

You can examine the execution history in the Execution History window, which you open by choosing **V**iew | **E**xecution History.

This window has two panes: the Instructions pane on top and the Keystroke Recording pane on the bottom.

The Instructions pane

The Instructions pane 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 84. As soon as you use the **Run** command or execute an interrupt, the execution history is deleted. (It starts being recorded again as soon as 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 Instructions pane local menu

The local menu for the Instructions pane contains three instructions:

```
Inspect
Reverse execute
------------------
Full history   Yes
```

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

The hot key for this command is 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. In this case, if you want to recover, you can use the keystroke replay facility of the Execution History window to reload your program and run forward to that point.

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 be on the lookout 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.

The Keystroke Recording pane

Even if you do inadvertently destroy your execution history, you can quickly execute back to a given point in your program, if you have *keystroke recording* enabled.

Keystroke recording works in conjunction with the reverse program execution capability to give you different ways of recovering to a previous point in your debugging session. It keeps a record of all the keys that you press, both when you're issuing commands to Turbo Debugger and when you're interacting with the program you are debugging. The keystrokes are recorded in a file named PROGNAME.TDK, where *progname* is the name of the program you are debugging.

Use the bottom pane of the Execution History window to replay keystrokes and recover to a previous point in your session. Each line in the keystroke history list shows the reason that Turbo Debugger gained control (breakpoint, trace, and so forth) and your program's current location at that time. If the location corresponds to a line of source code, that line is displayed. Otherwise, the instruction at that address is disassembled.

The **-k** command-line option enables keystroke recording. (See page 65.) You can also use TDINST to set the default to *On*.

The Keystroke Recording pane local menu

The local menu for the Keystroke Recording pane contains two commands: **I**nspect and **K**eystroke Restore.

Inspect

```
Inspect
Keystroke restore
```

If you highlight a line in the Keystroke Recording pane, then choose Inspect from the local menu, the Module window comes up with the cursor on the line of source code at which that keystroke occurred.

If this line does not correspond to a source code position, the CPU window opens with the highlight positioned on the instruction.

Keystroke Restore

If you highlight a line in the Keystroke Recording pane, then choose Keystroke Restore, Turbo Debugger reloads your program and runs it to the highlighted context. This is especially useful if you have executed a Turbo Debugger command that has deleted your execution history.

Interrupting program execution

With interactive programs, the quickest way to get to a specific place in your program is sometimes to simply run it, interact with it until it gets to the desired part of the code, and then interrupt execution. This is particularly true if the piece of code you want to examine is called several times before the one time of particular interest to you.

You may also want to interrupt program execution when, for some unexpected reason, control does not return to the debugger. This can happen when a piece of code contains an infinite loop: You expect a piece of code to be executed, so you set a breakpoint, but the breakpoint is never reached.

Ctrl-Break

This key combination will almost always interrupt your program and return control to the debugger. It takes effect as soon as you press it, so you might sometimes appear to be in an unexpected piece of code. This code could be in the ROM keyboard BIOS if your program is waiting for a keystroke, or at any instruction in the loop being executed. *Ctrl-Break* is unable to override the following two conditions—if either of these conditions occur, you will need to reboot your system:

- You are stuck in a loop with interrupts disabled.
- The system has crashed due to execution of erroneous code.

If you are debugging a program that needs to act upon the *Ctrl-Break* key combination itself, you can change the interrupt key. Use the TDINST installation program. You can set the interrupt key to be any key combination.

Program termination

When your program terminates and exits back to DOS, Turbo Debugger regains control. It displays a message showing the exit code that your program returned to DOS. Once your program terminates, you cannot use any of the **Run** menu options until you reload the program with **Run | Program Reset**.

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

Turbo Debugger has several features that make 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, Turbo Debugger lets you restart debugging but suspends execution before the last few commands that caused you to miss the problem that you wanted to observe.

Most debuggers force you to type in manually what could be a very long sequence of commands to get back to the place where the error occurred. Turbo Debugger has the powerful capability to record the keystrokes that made up the last session and to replay them on demand. It also lets you reload your last program from disk, with its previous DOS command-line arguments.

Reloading your program

To reload the program you were debugging, choose **Run | Program Reset**. Turbo Debugger reloads the program from disk, with any data you have added since you last saved to disk. This 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. Note that *Program Reset leaves breakpoints and watchpoints intact.*

Keystroke macro recording and playback

You can use the keystroke macro facility to record keystroke sequences that you use frequently. During debugging, you often repeat the same sequence of commands to get to a certain place in your program. This can be very tedious.

To get around this problem, you can define a keystroke macro that records all the keys you press, from when you first start Turbo Debugger until you have your program in the desired state. At that point, you can stop recording keystrokes. If you have to get back to the same place in your program, all you have to do is replay the keystroke macro.

☞ You can't use this utility to record keystrokes that must be typed to your program. You can only record Turbo Debugger command keystrokes.

The first thing you must do after starting Turbo Debugger from DOS is define a keystroke macro. Choose **O**ptions | **M**acros | **C**reate to do this. You're prompted to press a key to assign the keystroke macro to. Choose a key that hasn't been assigned to a function yet, such as *Shift* and one of the function keys, say *Shift-F1.* Now take your program to its point of crashing.

At that point, stop recording the keystroke macro by choosing **O**ptions | **M**acros | **S**top Recording. Save the macro to disk by choosing the **O**ptions | **S**ave Options command and turning on the Macros option in the Save Configuration dialog box. Continue running your program. After your program crashes, and you have reloaded it and Turbo Debugger, you can simply press *Shift-F1* to restart the program.

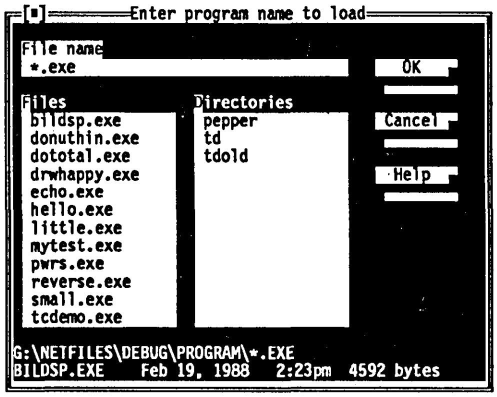
If your program requires you to type things to get to the next part of the recorded command sequence, you still have to enter those keystrokes manually. (You can do this while the macro is running.) For programs that do not require you to enter anything, this keystroke recording mechanism can completely automate the restarting procedure, saving many keystrokes.

☞ When a macro is saved to a configuration file, the configuration of the total environment is saved, including opened and zoomed windows. Thus if you record a macro that opens a window and don't close the window before saving the macro, the next time you restore that configuration file, the window will be open automatically even though you haven't executed the macro.

Opening a new program to debug

You load a new program to debug by choosing **F**ile | **O**pen to open the Load Program dialog box.

Figure 5.5 The Load Program dialog box



```
bildsp.exe        pepper
donuthin.exe      td
dototal.exe       tdold
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    Feb 19, 1988    2:23pm  4592 bytes
```

You can enter a file name (extension .EXE) in the File Name input box, or press *Enter* to active 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.*

If, instead, you type in the name of the file you want to load, the highlight bar 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, exactly as you would at the DOS prompt:

```
myprog a b c
```

This 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 DOS command line.

Once you have entered new arguments, Turbo Debugger 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

Turbo Debugger 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 the language you're using (C, Pascal, or assembler). If you are fairly new to a language and have not yet explored all its data types (**char**, **int**, integer, Boolean, real, single- and double-precision floating point, string, long integer, and so on), this chapter can still give you valuable information about them. When you have delved into the more complex data types (arrays, pointers, records, structures, unions, and so on), return to this chapter to learn more about looking at them with Turbo Debugger.

For how to examine or modify arbitrary blocks of memory as hex data bytes, see Chapter 11.

In this chapter, we show you how to examine and modify variables in your program. First, we explain the **D**ata menu and its options. We then discuss how you can modify program data by evaluating expressions that have side effects, and 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 of each language appear in Inspector windows.

The Data menu

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

The **D**ata 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 at the cursor, if any. 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 structures 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.

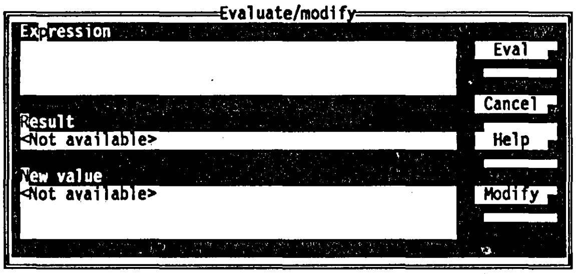
Evaluate/ Modify...

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 at the cursor, if any. If you select an expression (using *Ins*), the dialog box contains the marked expression.

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 that you want to watch. Turbo Debugger 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. This is useful when you want to watch something but have it displayed in a format other than Turbo Debugger's default display format for the data type.

The dialog box has three fields. You type the expression you want to evaluate in the top one. This 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 are debugging a C++ or object-oriented Pascal program, the Evaluate/Modify dialog box also lets you display the fields of an object instance or the members of a class 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 or member function, Turbo Debugger knows about the scope and presence of the *Self/this* parameter. You can evaluate *Self/this* and follow it with format specifiers and qualifiers.

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

Turbo Debugger also lets you call a method or member function from inside the Evaluate/Modify dialog box. Just type the instance name followed by a dot, followed by the method or member function 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 Turbo Debugger'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*

C programmers

The C language has a feature called *expressions with side effects* that can be powerful and convenient, as well as a source of surprises and confusion.

An expression with side effects alters the value of one or more variables or memory areas when it is evaluated. For example, the C increment (**++**) and decrement (**- -**) operators and the assignment operators (**=**, **+=**, and so on) have this effect. If you execute functions in your program within a C expression (for example, **myfunc(2)**), note that your function can have unexpected side effects.

If you don't intend to modify the value of any variable but merely want to evaluate an expression containing some of your program variables, don't use any of the operators that have side effects. On the other hand, side effects can be a quick and easy way to change the value of a variable or memory area. For example, to add 1 to the value of your variable named *count*, evaluate the C expression *count++*.

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 function is about to return. Use this command only when the function 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.

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

Pointing at data objects in source files

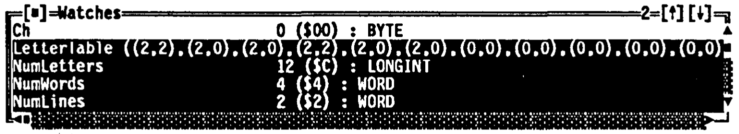
See Chapter 8 for a full discussion of using Module windows.

Turbo Debugger 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 the Inspect command.

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



```
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 | 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 structures appear with their values between braces ({ }) for C programs, and between parentheses for Pascal programs. 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 be watched, feel free to use variable names that are not yet valid because they are in a function that has not yet been called. This lets you set up a watch expression before its scope becomes active. This is the only situation in Turbo Debugger where you can enter an expression that cannot be immediately evaluated.

Warning!

This means that if you mistype the name of a variable, the mistake won't be detected because Turbo Debugger assumes it is the name of a variable that becomes available as your program executes.

Unless you use the scope-overriding mechanism discussed in Chapter 9, Turbo Debugger evaluates expressions in the Watches window in the scope of the current location where your program is stopped. Hence an expression in the Watches window is evaluated as if it appeared in your program at the place where the program is stopped. 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/this* parameter to the Watches window.

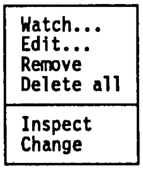
The Watches window local menu

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...

Prompts you for the variable name or expression to add to the Watches window. It is added to the beginning of the list.

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, record, or structure), this lets you view all its elements, not just the ones that fit in the Watches window. (The section "Inspector windows" on page 102 explains all about Inspector windows.)

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 are using permits it, Turbo Debugger performs any necessary type conversion exactly as if the appropriate assignment operator (= or :=) 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 **int**), pointers (**char** * in C, ^ in Pascal), arrays (**long** x[4], **array** [1..10] **of** Word), functions, structures, records, unions, and sets.

The Inspector window lists the items that make up the data object being inspected. The title of the window shows the data type of the inspected data and its name, if there is one.

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 an Inspector window as raw data bytes, select the View | Dump command 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 each of the languages supported by Turbo Debugger: C, 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.

C data Inspector windows

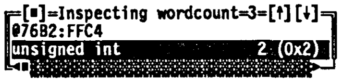
Scalars

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

```
char x = 4;
unsigned long y = 123456L;
```

Following the top line, these Inspector windows have only a single line of information that gives the address of the variable. To the left on the following line appears the type of the scalar variable (**char, unsigned long**, 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 C hex prefix of 0x). Use TDINST to change how the value is displayed.

If the variable being displayed is of type **char**, the equivalent character is also displayed. If the present value does not have a printing character equivalent, use the backslash (\) followed by a hex value to display the character value. This character value appears before the decimal or hex values.



```
[■]=Inspecting wordcount=3=[↑][↓]
@76B2:FFC4
unsigned int                2 (0x2)
```

Figure 6.3
A C scalar Inspector window

Pointers

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

```
char *p = "abc";
int  *ip = 0;
int  **ipp = &ip;
```

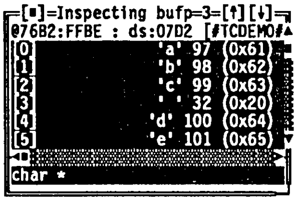
Pointer Inspector windows usually have a top line that contains the address of the variable, followed by a single line of information about the data pointed to. To the left appears [0], indicating the first member of an array. To the right appears the value of the item being pointed to. If the value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

If the pointer is of type **char** 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 then use the **Range** local menu command. This is an important technique for C programmers who use pointers to point to arrays of items as well as single items. For example, if you had the code

```
int array[10];
int *arrayp = array;
```

and you wanted to look at what *arrayp* pointed to, use the **Range** local command on *arrayp*, specifying a start index of 0 and a range of 10. If you had not done this, you would only have seen the first item in the array.



```
[■]=Inspecting bufp=3=[↑][↓]
@76B2:FFBE : ds:07D2 [#TCDEMO#
[0]                'a' 97 (0x61)
[1]                'b' 98 (0x62)
[2]                'c' 99 (0x63)
[3]                ' ' 32 (0x20)
[4]               'd' 100 (0x64)
[5]               'e' 101 (0x65)
char *
```

Figure 6.4
A C pointer Inspector window

Pointer Inspector windows also have a lower pane indicating the data type to which the pointer points.

Arrays

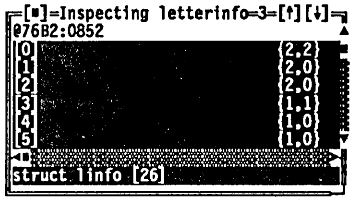
Array Inspector windows show you the value of arrays of data items, such as

```
long thread[3][4][5];
char message[] = "eat these words";
```

There is a line for each member of the array. To the left on each line appears the array index of the item. To the right appears the value of the item. If the value is a complex data item such as a structure or array, as much of it as possible is displayed.

You can use the **Range** local menu 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.

Figure 6.5
A C array Inspector window



```
[■]=Inspecting letterinfo=3=[↑][↓]
@76B2:0852
[0]                            {2,2}
[1]                            {2,0}
[2]                            {2,0}
[3]                            {1,1}
[4]                            {1,0}
[5]                            {1,0}
struct linfo [26]
```

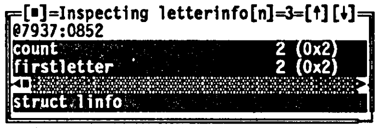
Structures and unions

Structure and union Inspector windows show you the value of the members in your structure and union data items. For example,

```
struct  date {
   int  year;
   char month;
   char day;
} today;

union {
   int  small;
   long large;
} holder;
```

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

Figure 6.6
A C structure or union
Inspector window

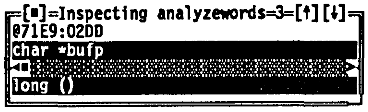


```
[■]=Inspecting letterinfo[n]=3=[↑][↓]
@7937:0852
count                          2 (0x2)
firstletter                    2 (0x2)
struct linfo
```

Structures and unions appear the same in Inspector windows. The lower pane of the Inspector window tells you whether you are looking at a structure or a union. These Inspector windows have as many items after the address as there are members in the structure or union. Each item shows the name of the member on the left and its value on the right, displayed in a format appropriate to its C data type.

Functions

Function Inspector windows show each parameter that a function is called with, below the memory address at the top of the window.



```
[■]=Inspecting analyzewords=3=[↑][↓]
@71E9:02DD
char *bufp
long ()
```

Figure 6.7
A C function Inspector window

They also give you information about the calling parameters, return data type, and calling conventions for a function. The lower pane indicates the data type returned by the function.

Pascal data Inspector windows

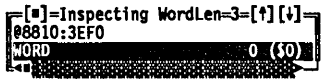
Scalars

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 TDINST 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, use 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 6.8
A Pascal scalar Inspector window

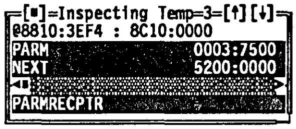
Pointers

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.

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



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

Figure 6.9
A Pascal pointer Inspector window

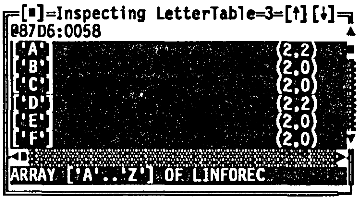
Arrays

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 **R**ange 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 6.10
A Pascal array Inspector window

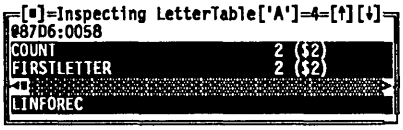
Records

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.

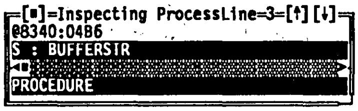


```
[■]=Inspecting LetterTable['A']=4=[↑][↓]
@87D6:0058
COUNT                          2 ($2)
FIRSTLETTER                    2 ($2)
LINFOREC
```

Figure 6.11
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, as well as the data type returned by a function.



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

Figure 6.12
A Pascal procedure Inspector window

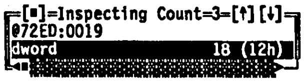
Assembler data Inspector windows

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 TDINST to change how the value is displayed.



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

Figure 6.13
An assembler scalar Inspector window

Pointers

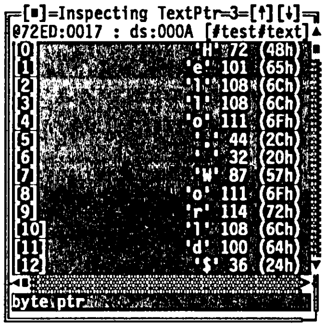
Pointer Inspector windows in assembler programs show you the value of data items that point to other data items, such as

```
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 a **STRUC** or array, however, only as much of it as possible is displayed, with the values enclosed in braces ({ and }).

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 **R**ange local 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 6.14
An assembler pointer Inspector window

Arrays

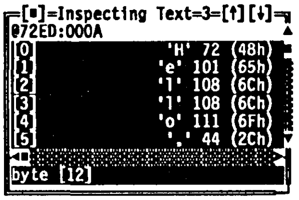
Array Inspector windows in assembler programs show you the value of arrays of data items, such as

```
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 **R**ange 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.

Figure 6.15
An assembler array Inspector window



```
=[■]=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]
```

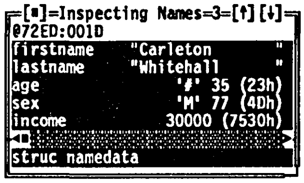
Structures and unions

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>
```

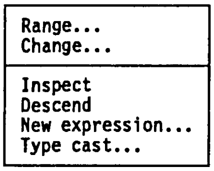
These Inspector windows have another pane below the one that shows the values of the fields. This additional pane shows the data type of the field highlighted in the top pane.

Figure 6.16
An assembler structure Inspector window



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

The Inspector window local menu



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 and inspect 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.

This command is particularly useful in C where you often declare a pointer to a data item—like **char** **p*—but what you really mean is that *p* points to an array of characters, not just a single character.

Change...

Changes the value of the currently highlighted item to the value you enter in the dialog box. If the current language permits it, Turbo Debugger 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 function, issuing the Inspect command shows you the source code for that function.

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

You can 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*, or click the close box with your mouse.

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 and C++ classes are somewhat different from regular Inspector windows. See Chapter 10 for a description of object type/class Inspector windows.

Type Cast...

Lets you specify a different data type (Byte, Word, Int, Char pointer) for the item being inspected. This 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

Turbo Debugger uses the single term "breakpoint" to refer to the debugger functions usually called 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.

Turbo Debugger 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
- *what happens* when the breakpoint is triggered

The location can be at either a single or global location in your program (if it is global, the breakpoint can occur at any source line or instruction in your program).

The "condition" can be

- always
- when an expression is true
- when a data object changes value

A "pass count" can also be specified, which requires "condition" to be true a certain number of times before the breakpoint can be triggered.

The "what happens" can be one of these:

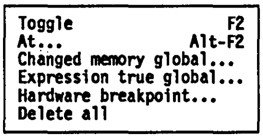
- stop program execution (a breakpoint)
- log the value of an expression
- execute an expression (splice code)

In this chapter, we'll show you how Turbo Debugger breakpoints give you more power and flexibility than traditional breakpoints, watchpoints, and tracepoints. You'll learn about the Breakpoints 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.

Many times, you just want to set a few simple breakpoints, so that if your program reaches any one of these locations, it stops. 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. Or, if you have a mouse, just click the first two columns of the line where you want to set the breakpoint (If you're in the right column, a ☼ appears in the position indicator). There is no limit to the number of breakpoints you can set.

The Breakpoints menu

You can access the **B**reakpoints 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

Sets or clears a breakpoint at the currently highlighted address in a Module window or CPU window Code pane. The hot key is *F2*.

At...

Lets you set a breakpoint at a specific location in your program. It opens a dialog box in which you can set all breakpoint options. *Alt-F2* is the hot key.

Changed Memory Global...

Sets a breakpoint that's triggered when an area of memory changes value. You are prompted for the area of memory to watch. For more information, see the **C**hanged Memory command in "The Breakpoints window local menu" section later in this chapter.

Expression True Global...

Sets a breakpoint that is triggered when the value of an expression you supply becomes true. You are prompted for the expression. For more information, see the **C**ondition Expression True command in "The Breakpoints window local menu" section later in this chapter.

Hardware Breakpoint...

Information on the hardware debugger interface is available in a file on your distribution disks. Refer to the README file for how to access this disk-based documentation.

Warning! You must have a hardware debugging board in order to use hardware debugging.

Delete All — Removes all the breakpoints you have set.

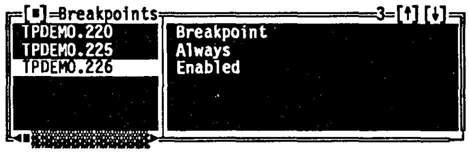
Scope of breakpoint expressions

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. See Chapter 9 for a complete discussion of scopes.

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.

The Breakpoints window

You open a Breakpoints window by choosing the **View | Breakpoints** command. This gives you a way of looking at and adjusting the conditions that trigger a breakpoint. You can use this window to add new breakpoints, delete breakpoints, and adjust existing breakpoints.



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

Figure 7.1
The Breakpoints window

Breakpoints windows have two panes. The left pane (Breakpoint List) shows a list of all the addresses at which breakpoints are set. The right pane (Breakpoint Detail) shows the details of the currently highlighted breakpoint in the left pane. Only the breakpoint list pane has a local menu, which you get to by pressing *Alt-F10*. Its options affect whatever breakpoint is highlighted in the Breakpoint List pane.

The Breakpoints window local menu

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

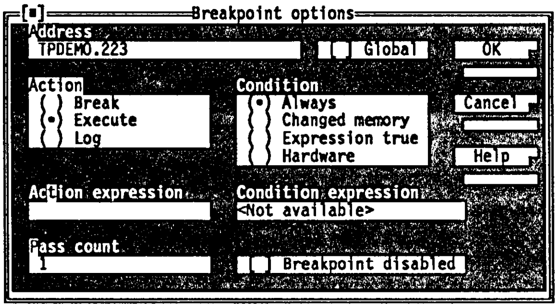
```
Set options...
Hardware options...
───────────────────
Add...
Remove
Delete all
Inspect
```

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

Set Options...

Opens the Breakpoint Options dialog box, which contains two sets of radio buttons, one input box, and one check box. In this dialog box, you can

- define what happens when the breakpoint highlighted in the Breakpoints List pane is triggered
- control the conditions under which the breakpoint is triggered
- set the number of times an action is encountered before the breakpoint triggers
- enable or disable the breakpoint
- set or change the breakpoint address
- make the breakpoint global



```
[■]══════════════Breakpoint options══════════════
Address
 TPDEMO.223                    [ ] Global     OK
Action               Condition
 ( ) Break            (•) Always              Cancel
 (•) Execute          ( ) Changed memory
 ( ) Log              ( ) Expression true     Help
                      ( ) Hardware
Action expression    Condition expression
                     <Not available>
Pass count
 1                    [ ] Breakpoint disabled
```

Figure 7.2
The Breakpoint Options dialog box

The Action radio buttons have three settings:

Break Causes your program to stop when the breakpoint is triggered. The Turbo Debugger screen reappears, and you can once again enter commands to look around at your program's data structures.

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. This option can act as a "code splice," letting you insert an expression that will execute before the code in your program at the current line number.

Log 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. See Chapter 9 for a description of expressions and side effects.

The Condition radio buttons have four settings:

Always Indicates that no additional conditions need be true before the breakpoint is triggered.

Changed Memory Watches a memory variable or object and allows the breakpoint to be triggered if the object changes. Use the Condition Expression input box to enter an expression reproducing the object you want to watch, followed by the number of objects to watch. The total number of bytes in the memory area is the size of the object that the expression references times the number of objects. For example, if you used C to enter

```
(long)a,4
```

the area watched for change would be 16 bytes long, since a **long** is 4 bytes and you said to watch four of them.

If you attach this condition to a global breakpoint, your program executes much more slowly because the memory area will have to be checked for change after every source line has been executed. If you've installed a hardware debugger device driver, changed memory breakpoints may become much faster. If a changed memory breakpoint has hardware assistance, an asterisk (*) appears after the breakpoint name in the left pane. You can expect then that the breakpoint will not slow down your program's execution.

By setting this condition on a breakpoint at a specific address, you do not incur the speed penalty of the global breakpoint, and you can still check the variable each time a specific line of code is executed.

Expression True Allows the breakpoint to be triggered when an expression becomes true (nonzero). Use the Condition Expression input box to enter an expression to evaluate each time the action is encountered.

See disk-based documentation about the hardware debugger interface and the options available under this menu.

Hardware Causes the breakpoint to be triggered by the hardware-assisted device driver. Use this menu either if you have a 386 system and are using the TDH386.SYS device driver, or if you have a hardware debugger board installed in your system and the board vendor supplies a Turbo Debugger device driver.

The Pass Count input box lets you set the number of times the breakpoint action must occur before the breakpoint is triggered. The default number is 1. The pass count is decremented only when the condition attached to the breakpoint is true. This means that if you set a pass count as well as a condition, it causes the breakpoint to be triggered the *n*th time that the condition is true.

The Breakpoint Disabled check box lets you enable or disable the currently highlighted breakpoint. A disabled breakpoint is "invisible" until you enable it again; it behaves as if it had been deleted.

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 re-enter it along with its conditions and action.

Hardware Options... Refer to the disk-based documentation about the hardware debugger interface for how to use this option.

Warning! You must have a hardware debugging board in order to use hardware debugging.

Add... Opens a dialog box like the Set Options dialog box. You must enter an address in the Address input box.

You can also add a breakpoint by simply starting to type the address at which you want to set it. A dialog box appears just as if you had invoked the **Add** command.

Once you've added the breakpoint, you can use the other local menu commands to modify its behavior. When you first add a breakpoint, it has a pass count of 1, its condition is set to always occur, and the action is to break (stop) your program.

Remove — Removes the currently highlighted breakpoint.

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.

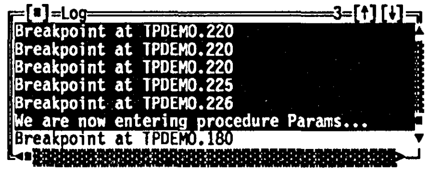
Inspect — Shows you the source code line or assembler instruction that corresponds to the currently highlighted breakpoint item. If the breakpoint is set at an address that corresponds to a source line in your program, a Module window is opened and set to that line. Otherwise, a CPU window is opened, 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.

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.

Figure 7.3
The Log window



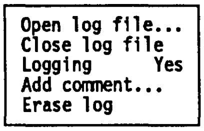
```
Breakpoint at TPDEMO.220
Breakpoint at TPDEMO.220
Breakpoint at TPDEMO.220
Breakpoint at TPDEMO.225
Breakpoint at TPDEMO.226
We are now entering procedure Params...
Breakpoint at TPDEMO.180
```

Log windows show 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. To adjust the number of lines, use either a command-line option at startup or permanently change the number using the TDINST customization program. You can preserve the entire log, continuously writing it to a disk file, by using the **O**pen 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 **A**dd 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 **W**indow | **D**ump Pane to Log command (from the menu bar) to record the current contents of a pane in a window.

The Log window local menu



```
Open log file...
Close log file
Logging      Yes
Add comment...
Erase log
```

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, and clearing 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 desired 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 **E**rase Log File before choosing **O**pen Log File.

Close Log File

Stops writing lines to the log file specified in the **O**pen Log File local menu command, and the file is closed.

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 desire.

Erase Log Clears the log list. The Log window will now be blank. This does not affect writing the log to a disk file.

Simple breakpoints

One of the most common things you'll want to do during debugging is cause your program to stop if certain pieces of code are about to be executed.

There are a number of ways to set a breakpoint. Each one is convenient in different circumstances:

- Move to the desired source line in a Module window and issue the **B**reakpoints | **T**oggle command (or press *F2* or click the line with your mouse). Doing this on a line that already has a breakpoint set causes that breakpoint to be deleted.
- Move to an instruction in the Code pane of a CPU window and issue the **B**reakpoints | **T**oggle command (or press *F2* or click the line with your mouse). Doing this on a line that already has a breakpoint set causes that breakpoint to be deleted.
- Issue the **B**reakpoints | **A**t command and enter a code address at which to set a breakpoint. (A code address has the same format as a pointer in the current language. See Chapter 9 about expressions.)
- Issue the **A**dd local menu command from the Breakpoint List pane of the Breakpoints window and enter a code address at which to set a breakpoint.

Conditional breakpoints and pass counts

There are many occasions where you do not 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. Turbo Debugger gives you two ways to qualify when a breakpoint is actually triggered: *pass counts* and *conditions*.

If you want to stop your program on the tenth call to a function, you can set a breakpoint at the start of the function and use the Pass Count input box in the Breakpoint Options dialog box to set the number of times you want to skip the breakpoint before it is actually triggered.

If you want to stop your program at a specific location but only when a certain condition is true, you can specify an expression using the Expression True radio button in the Breakpoint Options dialog box. Each time the breakpoint is encountered, the expression will be evaluated, and if it is true (nonzero), the breakpoint will be triggered. This can be used in combination with the pass count to trigger a breakpoint only after the expression has been true a certain number of times.

You can use the Changed Memory radio button to specify a breakpoint that occurs only after a data item changes value. This can be a lot more efficient than specifying a global breakpoint that watches for exactly when something changes. If you only watch for something to change when a specific source statement is reached, it reduces the amount of processing Turbo Debugger does in order to detect when the change occurred.

Global breakpoints

If you want to have a breakpoint occur every time a source line or instruction is encountered, use global breakpoints. There are a number of ways to create a global breakpoint, each best-suited for a particular situation:

- In the Breakpoint Options dialog box, turn on the Global check box. Use this method when you want to set a qualifying condition or pass count, or when you want to do something other than stop when the breakpoint is triggered.
- Choose the **B**reakpoints | **C**hanged Memory Global command to stop when a specific area of memory changes.
- Choose the **B**reakpoints | **E**xpression True Global command to stop execution when an expression becomes true.

When you set a global breakpoint, you usually use the local menu in the Breakpoints window to modify the condition or the action; otherwise, all you end up with is a breakpoint action that occurs on every source line—just like using the **R**un | **T**race Into main menu command.

If you want to test your global breakpoints each time a source line is about to be executed, make sure your current window is not a CPU window, then restart your program with one of the **Run** commands from the menu bar (or its function-key equivalents).

To test your global actions each time a single instruction is executed, make sure your current window is a CPU window when you restart your program.

Warning! A global action will occur on every source line or instruction. Use a global breakpoint when you want to find out exactly when a variable changes or when some condition becomes true.

Global breakpoints greatly slow the execution of your program. However, they can be very convenient for finding where your program is "bashing" data.

☞ After adding the global breakpoint, you *must* set a condition that will trigger it.

Breaking for changed data objects

When you want to find out where in your program a certain data object is being changed, first set a global breakpoint using one of the techniques outlined in the previous section. Then use the Changed Memory radio button in the Breakpoint Options dialog box. When the input box appears, enter an expression that refers to the memory area you want to keep track of, along with an optional count of the number of objects to track.

Your program will execute slowly when you use this command. You may want to localize the problem before using this technique to find the exact location where a data item changes.

If you have installed a hardware device driver, Turbo Debugger 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 has used the hardware by opening a Breakpoint window with the **V**iew | **B**reakpoints command. Any breakpoint that is hardware assisted will have an asterisk (*) beside it. These breakpoints will be much faster than global breakpoints that are not hardware assisted.

Logging variable values

You can only set one breakpoint per address.

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 a function is called with. By looking at the log each time the function is called, you can determine when it was called with erroneous parameters.

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. If you want to log the value of multiple variables, you must set multiple breakpoints.

Executing expressions

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.

C H A P T E R

8

Examining and modifying files

Turbo Debugger treats disk files as a natural extension of the program you're debugging. You can examine and modify any file on the disk, viewing it either as ASCII text or as hex data. You can also make changes to text files using your favorite word processor or text editor, all from within Turbo Debugger.

This chapter shows you how to examine and modify two sorts of disk files: those that contain your program source code, and other files on disk.

Examining program source files

Program source files are your source files that are compiled and that 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 function to verify either that its arguments are valid or that it is returning a correct value.

As you step through your program, Turbo Debugger automatically displays the source code for the current location in your program.

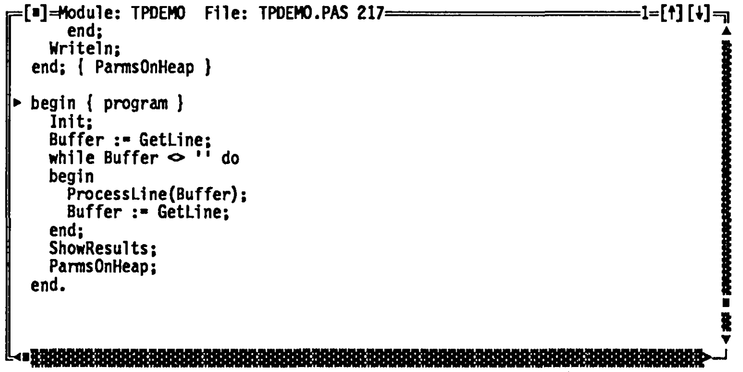
Files that are included in a source file by a compiler directive and generate line #s (like **#include** in C and **INCLUDE** in assembler) are also considered to be program source files (that is, when you choose **V**iew | **M**odule, they appear in the Pick a Module list pane).

You should always use a Module window to look at your program source files because this informs Turbo Debugger that the file is a source module. It can then let 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

You create a Module window by choosing the **V**iew | **M**odule command from the menu bar (or pressing the hot key, *F3*).

Figure 8.1
The Module window



```
=[■]=Module: TPDEMO  File: TPDEMO.PAS 217=====================1=[↑][↓]=
      end;
    Writeln;
  end; { ParmsOnHeap }

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

A dialog box appears in which you can enter the name of the module you want to view.

Turbo Debugger will then load the source file for the module that you select. It searches for the source file in the following places:

1. in the directory where the compiler found the .EXE file
2. in the directories specified by the **O**ptions | **P**ath 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).

Note that when you run Turbo Debugger, you'll need *both* the .EXE file and the original source file. Turbo Debugger searches for source files first in the directory the compiler found them in when it compiled, second in the directory specified in the **O**ptions | **P**ath for Source command, third in the current directory, and fourth in the directory the .EXE file is in.

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. This means that 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. This can cause the arrow that shows the current program location (CS:IP) to be displayed on the wrong line.

The Module window local menu

```
Inspect
Watch
----------
Module...
File...
----------
Previous
Line...
Search...
Next
Origin
Goto...
Edit
```

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 Turbo Debugger, so you should become quite familiar with its various options.

Use the *Alt-F10* key combination to pop up the Module window local menu or, if you have control-key shortcuts enabled, use the *Ctrl* key with the first letter of the desired command to access that command (for example, *Ctrl-S* for **S**earch).

Inspect

Opens an Inspector window to show you the contents of the program variable at the current cursor position. Before issuing this command, you can place the cursor at the program variables in the source file that you want to inspect, or you can enter it in the input box of the dialog box that appears.

You can also use the *Ins* key to select (highlight) an expression to inspect. This saves you from typing in an expression that is in plain view in the source module.

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 — Adds the variable at the current cursor position to the Watches window. This is useful if you want to monitor the value of a variable continuously as your program executes. Before issuing this command, you can place the cursor at the program variables in the source file that you want to inspect, or you can enter it in the input box of the dialog box that appears.

You can also use the *Ins* key to mark an expression to watch. This saves you from typing in an expression that is in plain view in the source module.

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 | File** to look at the first file. If you want to see more than one, use **View | Another | File** to open subsequent File 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 last line in 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 **S**earch command; you can only use this after issuing a **S**earch command.

Sometimes, a search command matches an unexpected string before reaching the one you really wanted to find. **N**ext 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 line number, a function name, or a hex address. See Chapter 9 for a complete description of the ways to enter an address.

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 specified the **Run** | **E**xecute To command. This is a handy hot key for this frequently used command.

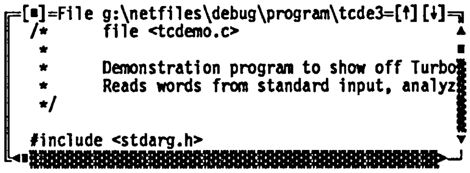
Edit Starts up your choice of an editor so that you can make changes to the source file for the module you are viewing. You can specify the command that starts your editor from the installation program TDINST.

Examining other disk files

You can examine or modify 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 **D**isplay As command described in a later section of this chapter.

The File window

You create a File window by choosing the **V**iew | **F**ile command 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.

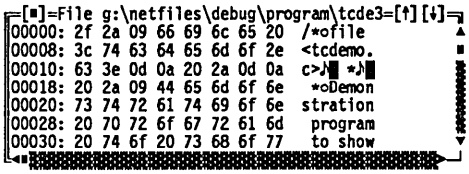


```
[■]=File g:\netfiles\debug\program\tcde3=[↑][↓]
/*      file <tcdemo.c>
 *
 *      Demonstration program to show off Turbo
 *      Reads words from standard input, analyz
 */

#include <stdarg.h>
```

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 will appear either as ASCII text or as hexadecimal bytes, depending on whether the file contains what Turbo Debugger thinks is ASCII text or binary data. You can switch between ASCII and hex display at any time using the **D**isplay As local menu command described later.

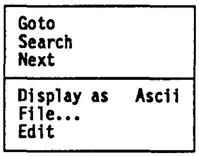


```
[■]=File g:\netfiles\debug\program\tcde3=[↑][↓]
00000: 2f 2a 09 66 69 6c 65 20  /*ofile
00008: 3c 74 63 64 65 6d 6f 2e  <tcdemo.
00010: 63 3e 0d 0a 20 2a 0d 0a  c>♪█ *♪█
00018: 20 2a 09 44 65 6d 6f 6e   *oDemon
00020: 73 74 72 61 74 69 6f 6e  stration
00028: 20 70 72 6f 67 72 61 6d   program
00030: 20 74 6f 20 73 68 6f 77   to show
```

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.



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 first letter of the desired command to access it.

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, you will be positioned 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 wildcards, with ? indicating a match on any single character, and * matching 0 or more characters.

See Chapter 9 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.

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 **S**earch command.

Next Searches for the next instance of the character string you specified with the **S**earch command; you can only use this command *after* first issuing a **S**earch command.

This is useful when your **S**earch 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 hexadecimal bytes. When you select **A**SCII display, the file appears as you are used to seeing it on the screen in an editor or word processor. If you select **H**ex 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-style wildcards to get a list of file choices, or you can type a specific file name to load. This 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, issue the **V**iew | **A**nother | **F**ile command to make another File window.

Edit If you are viewing the file as ASCII text, this command lets you make changes to the file you are viewing by invoking the editor you specified with the TDINST installation program.

Chapter 9 has a complete description of byte lists.

If you are viewing the file as hex data bytes, the debugger does not start your editor. Instead, you are prompted for the bytes to replace those at the current cursor position. Enter a byte list, just as if you were entering a list of bytes to search for.

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.

Turbo Debugger can evaluate expressions and tell you their value. You can also use expressions to indicate a data item 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 **D**ata | **E**valuate/Modify to open the Evaluate/Modify dialog box, which tells you the value of an expression. (You can also use this dialog box as a simple calculator.)

In this chapter, you'll learn how Turbo Debugger 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 Turbo Debugger supports and the syntax of expressions.

For a complete discussion of C, Pascal, and assembler expressions, refer to your *Turbo C Getting Started* and *Programmer's Guide,* the *Turbo Pascal User's Guide* and *Reference Guide,* or the *Turbo Assembler Reference Guide.*

Choosing the language for expression evaluation

Turbo Debugger 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, C, Pascal, or Assembler. If you choose Source, expressions are evaluated in the manner of the module's language. (If Turbo Debugger can't determine the module's language, it uses the expression rules for Turbo Assembler.)

Usually, you let Turbo Debugger 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 is convenient to treat expressions or variables as if they had been written in a different language; for example, if you are debugging a Pascal program, assembly language or C conventions may offer an easier way to change the value of a byte stored in a string.

So long as your initial choice of language is correct when you enter Turbo Debugger, you should have no difficulty using other language conventions. Turbo Debugger still retains information about the original source language and will handle the conversions and data storage appropriately. If the language seems ambiguous, Turbo Debugger defaults to assembly language.

Even if you deliberately choose the wrong language when you enter Turbo Debugger, 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 confuse Turbo Debugger into storing 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 evalutes to a memory pointer, or specify code addresses as source line numbers by preceding the line number with a pound sign (#), like #123. 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
C	0x*nnnn*	0x1234:0x0010
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 Turbo Debugger in most situations.

Normally, Turbo Debugger looks for a symbol in an expression the same way a compiler would. For example, C first looks in the current function, then in the current module for a static (local) symbol, then for a global symbol. 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.

If Turbo Debugger doesn't find a symbol using these techniques, it searches through all the other modules to find a static symbol that matches. This lets you reference identifiers in other modules without having to explicitly mention the module name.

If you want to force Turbo Debugger to look elsewhere for a symbol, you can exert total control over where to look for a symbol name by specifying a module, a file within a module, or a routine to look inside. You can access any symbol in your program that has a defined value, even symbols that are private to a function or procedure and have names that conflict with other symbols.

Scope override syntax

No matter what language you're using, you use the same method to override the scope of a symbol name.

Normally, you use a pound sign (#) to separate the components of the scope. If it's not ambiguous in the current language, you can also use a period (.) instead of # and omit the initial pound sign.

The following syntax describes scope overriding; brackets ([]) indicate optional items:

```
[#module[#filename]]#linenumber[#variablename]
```

or

```
[#module[#filename]][#functionname]#variablename
```

If you don't specify a module, the current module 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.

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#123`	Line 123 in source file *file1*, which is part of module *mymodule*
`#mymodule#file1#123#myvar1`	Symbol *myvar1* accessible from line 123 in source file *file1*, 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#myvar2`	Variable *myvar2* accessible from *file2,* which is included in *mymodule*
`#mymodule#file2#myfunc #myvar2`	Variable *myvar2* accessible from *myfunc* defined in file *file2,* which is included in *mymodule*

Turbo Debugger also supports Pascal's unit-override syntax:

```
unitname.symbolname
```

OOP

Finally, Turbo Debugger lets you override scope by using object, class, method, and member function names. Here's some examples:

`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 *AMethod* accessible in instance *AnInstance* accessible in unit *AUnit*
`AUnit.AnObjectType.AMethod`	Method *AMethod* accessible in object type *AnObjectType* accessible in unit *AUnit*

`AUnit.AnObjectType.AMethod.ANestedProc.AVar`	Local variable *AVar* accessible in procedure *ANestedProc* accessible in method *AMethod* accessible in object type *AnObjectType* accessible in unit *AUnit*

You can enter such qualified identifier expressions anywhere an expression is valid, including in the Evaluate/Modify dialog box and the Watches window, or when you're changing an expression in an Inspector window or using the local menu in the Module window to **G**oto a method, member function, or procedure address in the source code.

☞ If you are debugging a C++ program and want to examine a function with an *overloaded name*, just enter the name of the function in the appropriate input box. Turbo Debugger opens the Pick a Symbol Name dialog box with a list box of all the functions of that name, with their arguments, so you can choose the one you want.

Implied scope for expression evaluation

Whenever Turbo Debugger evaluates an expression, it must decide where in your program the "current scope" is that is used 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; Turbo Debugger must know which instance of a symbol you mean.

Turbo Debugger usually uses the current cursor position as the context for "deciding" about 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.

This means that if you have moved the cursor off the current line where your program is stopped, you may get unexpected results from evaluating expressions. If you want to be sure that expressions are evaluated in your program's current scope, use the **O**rigin 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 cursoring to a routine in the Stack window, or by cursoring to a routine name in a Variables window.

Byte lists

Several commands ask you to enter a list of bytes, including the **S**earch and **C**hange local menu commands in the Data pane of the CPU window, and the **S**earch and **C**hange local menu commands 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 **O**ptions | **L**anguage command. Both strings and scalars use the same syntax as expressions. Scalars are converted into a corresponding byte sequence. For example, a Pascal Longint value of 123456 becomes a 4-byte hex quantity 40 E2 01 00.

Language	Byte list	Hex data
C	"ab" 0x04 "c"	61 62 04 63
Pascal	'ab'#4'c'	61 62 04 63
Assembler	1234 "AB"	34 12 41 42

C expressions

Turbo Debugger supports the complete C expression syntax. A C expression consists of a mixture of symbols, operators, strings, variables, and constants. Each of these components is described in one of the following sections.

C symbols

A symbol is the name of a data object or routine in your program. A symbol name starts with a letter (*a-z*, *A-Z*) or underscore (_). Subsequent characters in the symbol may contain these characters and also the digits 0 through 9. You can omit the beginning underscore from symbol names; if you enter a symbol name without an underscore and that name cannot be found, it is searched for again with an underscore at the beginning. The compiler automatically puts an underscore at the start of your symbol names, which saves you from having to remember to add it.

C register pseudovariables

Turbo Debugger lets you access the processor registers using the same technique as the Turbo C compiler, namely pseudovariables. A *pseudovariable* is a variable name that corresponds to a given processor register.

Pseudovariable	Type	Register
_AX	unsigned int	AX
_AL	unsigned char	AL
_AH	unsigned char	AH
_BX	unsigned int	BX
_BL	unsigned char	BL
_BH	unsigned char	BH
_CX	unsigned int	CX
_CL	unsigned char	CL
_CH	unsigned char	CH
_DX	unsigned int	DX
_DL	unsigned char	DL
_DH	unsigned char	DH
_CS	unsigned int	CS
_DS	unsigned char	DS
_SS	unsigned char	SS
_ES	unsigned char	ES
_SP	unsigned int	SP
_BP	unsigned char	BP
_DI	unsigned char	DI
_SI	unsigned char	SI
_IP	unsigned int	IP

The following pseudovariables let you access the 80386 processor registers:

Pseudovariable	Type	Register
_EAX	unsigned long	EAX
_EBX	unsigned long	EBX
_ECX	unsigned long	ECX
_EDX	unsigned long	EDX
_ESP	unsigned long	ESP
_EBP	unsigned long	EBP
_EDI	unsigned long	EDI
_ESI	unsigned long	ESI
_FS	unsigned int	FS
_GS	unsigned int	GS

C constants and number formats

Constants can be either floating point or integer.

An integer constant is specified in decimal, unless one of the C conventions for overriding this is used:

Format	Radix
digits	decimal
0digits	octal
0Xdigits	hexadecimal
0xdigits	hexadecimal

Constants are normally of type **int** (16 bits). If you want to define a **long** (32-bit) constant, you must add an *l* or *L* at the end of the number. For example, *123456L*.

A floating-point constant contains a decimal point and can use decimal or scientific notation. For example,

```
1.234 4.5e+11
```

Escape sequences

A string is a sequence of characters enclosed in double quotes ("").

You can use the standard C backslash (\) as an escape character.

Sequence	Value	Character
\\	0X5C	Backslash
\a	0X07	Bell
\b	0X08	Backspace
\f	0X0C	Formfeed
\n	0X0A	Newline
\r	0X0D	Carriage return
\t	0X09	Horizontal tab
\v	0X0B	Vertical tab
\x*nn*	*nn*	Hex byte value
\nnn	*nnn*	Octal byte value

If you follow the backslash with any other character than those listed here, that character is inserted into the string unchanged.

C operators precedence

Turbo Debugger uses the same operators as C, with the same precedence. The debugger has one operator that is part of the C++ set of operators: the double colon (::). This operator has a higher priority than any of the regular C operators. It is used to make a constant far address out of the expression that precedes it and the expression that follows it; for example,

```
0X1234::0X1000
_ES::_BX
```

The primary expression operators

[]   .   ->   **sizeof**

have the highest priority, from left to right. The unary operators

*   &   –   !   ~   ++   --

are of a lower priority than the primary operators but a greater priority than the binary operators, grouped from right to left. The priority of the binary operators, in descending order, is as follows (operators on the same line have the same priority):

highest	* / %
	+ −
	>> <<
	< > <= >=
	== !=
	&
	^
	|
	&&
lowest	||

The single ternary operator, **?:**, has a priority below that of the binary operators.

The assignment operators are below the ternary operator in priority. They are all of equal priority, and group from right to left:

```
= += −= *= /= %= >>= <<= &= ^= |=
```

Executing C functions in your program

You can call functions from a C expression exactly as you do in your source code. Turbo Debugger actually executes your program code with the function arguments that you supply. This can be a very useful way of quickly testing the behavior of a function you've written. You can repeatedly call it with different arguments and then check that the returned value is correct each time.

The following function raises one integer number to a power (x^y):

```
long power(int x, int y)
{
   long temp = 1;
   while (y--)
      temp *= x;
   return(temp);
}
```

The following table shows the result of calls to this function with different function arguments:

C expression	**Result**
power(3,2) * 2	18
25 + power(5,8)	390650
power(2)	Error (missing argument)

C expressions with side effects

A side effect occurs when you evaluate a C expression that changes the value of a data item in the process of being evaluated. In some cases, you may want a side effect, using it to intentionally modify the value of a program variable. At other times, you want to be careful to avoid them, so it's important to understand when a side effect can occur.

The assignment operators (**=**, **+=**, and so on) change the value of the data item on the left side of the operator. The increment and decrement (**++** and **– –**) operators change the value of the data item that they precede or follow, depending on whether they are used as prefix or postfix operators.

A more subtle type of side effect can occur if you execute a function that's part of your program. For example, if you evaluate the C expression

```
myfunc(1,2,3) + 7
```

your program may misbehave later if **myfunc** changed the value of other variables in your program.

C reserved words and type conversion

Turbo Debugger lets you perform type conversions on (cast) pointers exactly as you would do in a C program. A *type conversion* consists of a C data-type declaration between parentheses. It must come before an expression that evaluates to a memory pointer.

Type conversions are useful if you want to examine the contents of a memory location pointed to by a far address you generated using the double colon (::) operator, for example,

```
(long far *)0x3456::0
(char far *)_ES::_BX
```

You can use a type conversion to access a program variable for which there is no type information, which happens when you compile a module without generating debugging-type information. Rather than recompiling and relinking, if you know the data type of a variable, you can simply put that in a type conversion before the name of the variable.

For example, if your variable *iptr* is a pointer to an integer, you can examine the integer that it points to by evaluating the C expression

```
*(int *)iptr
```

You can also use the **Type** Cast command in the Inspector window local menu for this purpose.

Use the following C reserved words to perform type conversions for Turbo Debugger:

char	**huge**	**struct**
double	**int**	**union**
enum	**long**	**unsigned**
far	**near**	
float	**short**	

Pascal expressions

Turbo Debugger 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.

Pascal 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 139.

Pascal 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.

Pascal 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'
```

Pascal operators and operator precedence

Turbo Debugger 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, as in C.

Calling Pascal functions and procedures

You can reference 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 the **Data | Evaluate/Modify** command 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, Turbo Debugger 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

Turbo Debugger 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

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

Turbo Debugger supports most of the assembler operators, listed here in order of 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, Turbo Debugger displays it in a format based on the type of data it is. Turbo Debugger 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. The string is terminated with a null character.
x or h	Displays an integer as a hexadecimal number.

C H A P T E R

10

C++ and object-oriented Pascal debugging

OOP

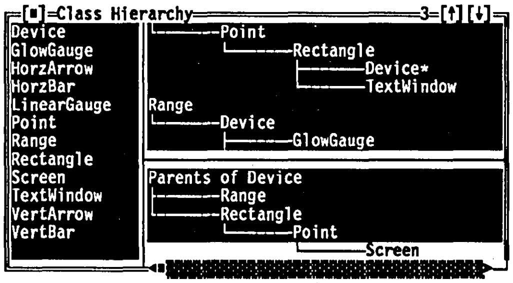
To meet the needs of the C++ and object-oriented Pascal revolution, Turbo Debugger has been enhanced to support object-oriented programming. To use these new features, you must have version 5.5 of Turbo Pascal or Turbo C++, and version 2.0 of Turbo Debugger.

Besides extensions that let you trace into object methods or class member functions and examine objects or classes in the Evaluate/ Modify dialog box and the Watches window, Turbo Debugger 2.0 comes equipped with a special set of windows and local menus specifically designed for objects and classes.

The Hierarchy window

Turbo Debugger provides a special window for examining object or class 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         └───────Device
Range                 ├──────GlowGauge
Rectangle
Screen        Parents of Device
TextWindow    ├───────Range
VertArrow     └───────Rectangle
VertBar               └──────Point
                             └──────Screen
```

Use Tab to move between the two panes.

The Hierarchy window displays information on object or class *types* rather than instances. The left pane lists in alphabetical order the types used by the module being debugged. The right pane (two panes if you are running a C++ program with multiple inheritance) shows all objects or classes in their hierarchies, using a line graphic that places the base type at the left margin of the pane and displays descendants (also ancestors for classes with multiple inheritance) beneath and to the right of the base type, with lines indicating ancestor and descendant relationships.

The Object Type List pane

The left pane provides an alphabetical list of all object or class types used by the current module. It supports an incremental matching feature to eliminate the need to cursor through large lists of types: When the highlight bar is in the left pane, simply start typing the name of the object or class type you're looking for. At each keypress, Turbo Debugger highlights the first type matching all keys pressed up to that point.

Press *Enter* to open an object type/class Inspector window for the highlighted type. Object type/class Inspector windows are described on page 158.

The Object Type/Class 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 TDINST. This local menu contains two items: Inspect and Tree.

```
Inspect
Tree
```

Inspect

Displays an object type/class Inspector window for the highlighted type.

Tree

Moves to the right pane of the window, 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 objects or classes 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 or class type in a complex hierarchy tree, go back to the left pane and use the incremental search feature; then choose the **Tree** item 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/class Inspector window appears for the highlighted type.

☞ If you have loaded a C++ program that uses classes with multiple inheritance, a third pane, the Parent Tree pane, appears below the Hierarchy Tree pane in the Hierarchy window. If the class you are examining has multiple ancestors, and if the **Parent** command in the Hierarchy Tree pane local menu is set to *Yes*, a reverse tree appears in the Parent Tree pane with the message `Parents of Class` at the left margin of the pane and the ancestors displayed beneath and to the right, with lines indicating descendant and ancestor relationships.

You can open an object type/class Inspector window for any class that appears in the Parent Tree pane, just as you can in the Hierarchy Tree pane.

The Hierarchy Tree pane local menu(s)

```
Inspect
```

The Hierarchy Tree pane local menu (*Alt-F10* in the right pane) has only one item: **Inspect**. When you choose it, an object type/class 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.

```
Inspect
Parents  Yes
```

If you have loaded a C++ program that uses classes with multiple inheritance, the Hierarchy Tree pane local menu contains a second command, **Parents**. This is a toggle with which you can control whether to show the ancestors of a class in the Parent Tree pane. This is useful if a class you are examining has multiple inheritance. The default for **Parents** is *Yes.*

The Parent Tree pane local menu

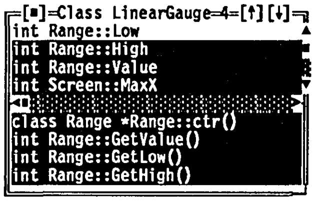
Inspect

Finally, the Parent Tree pane, if it exists, has a local menu of its own, with a single command, **Inspect**. It works just the same as the **Inspect** command in the Hierarchy Tree pane local menu: It opens an Inspector window for the highlighted object type or class.

Object type/class Inspector windows

Turbo Debugger provides a special type of Inspector window to let you inspect the details of an object type: the object type/class Inspector window. The window summarizes type information, but does not reference any particular instance.

Figure 10.2
An object type/class Inspector window



```
[■]=Class LinearGauge=4=[↑][↓]
int Range::Low
int Range::High
int Range::Value
int Screen::MaxX

class Range *Range::ctr()
int Range::GetValue()
int Range::GetLow()
int Range::GetHigh()
```

The window is divided horizontally into two panes, with the top pane listing the data fields or members of the type and the bottom pane listing the method or member function names and (if the selected item is a function rather than a procedure) the function return type. Use the *Tab* key to move between the two panes of the object type/class Inspector window.

If the highlighted data field is an object or class type, or a pointer to an object or class type, pressing *Enter* opens another object type/class Inspector window for the highlighted type. (This action is identical to selecting the **Inspect** command in the local menu for this pane.) In this way, complex nested structures of objects or classes can be inspected quickly with a minimum of keystrokes.

For brevity's sake, method or member function parameters are not shown in the object type/class Inspector window. To examine parameters, highlight the method or member function and press *Enter*. A method/member function Inspector window appears. The top pane of the window displays the code address for the object or class type's implementation of the selected method or member function, and the names and types of all its parameters. If your source program is in object-oriented Pascal, the bottom pane of the window indicates whether the method is a procedure or a function.

Pressing *Enter* from anywhere within the method/member function Inspector window brings the Module window to the foreground, with the cursor at the code that implements the method or member function being inspected.

As with standard inspectors, *Esc* closes the current Inspector window and *Alt-F3* closes them all.

The object type/class Inspector window local menus

Pressing *Alt-F10* brings up the local menu for either pane. If control-key shortcuts are enabled (through TDINST), 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 pane local menu contains these items:

Inspect

If the highlighted field is an object or class type or a pointer to one, a new object type/class Inspector window is opened for the highlighted field.

Hierarchy

Opens an Hierarchy window for the object or class type being inspected. The Hierarchy window is described on page 155.

Show Inherited

```
Inspect
Hierarchy
Show inherited   Yes
```

Yes is the default value of this toggle. When it is set to *Yes*, all data fields or members are shown, whether they are defined within the type of the inspected object or class or inherited from an ancestor type. When it is set to *No*, only those fields/members defined within the type being inspected are displayed.

The local menu commands for the bottom Object Method pane are Inspect, Hierarchy, and Show Inherited.

Inspect — A method/member function 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/member function Inspector window, the Module window is brought to the foreground with the cursor at the code that implements what is being inspected.

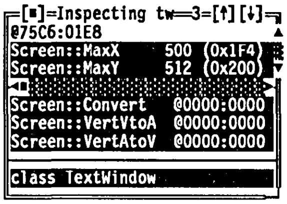
Hierarchy — Opens an Hierarchy window for the object or class type being inspected. The Hierarchy window is described on page 155.

Show Inherited — *Yes* is the default value of this toggle. When it is set to *Yes*, all methods or member functions 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 or member functions are displayed that are defined within the object type being inspected.

Object instance Inspector windows

Object type/class Inspector windows provide information about object or class types, but say nothing about the data contained in a particular object or class instance at a particular time during program execution. Turbo Debugger provides an extended form of the familiar record Inspector window specifically to inspect object and class instances.

Bring up this window by placing your cursor on an object or class instance in the Module window, then pressing *Ctrl-I*.



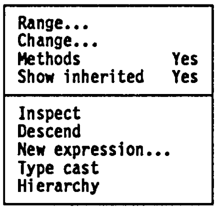
```
[■]=Inspecting tw=3=[↑][↓]
@75C6:01E8
Screen::MaxX       500 (0x1F4)
Screen::MaxY       512 (0x200)

Screen::Convert    @0000:0000
Screen::VertVtoA   @0000:0000
Screen::VertAtoV   @0000:0000

class TextWindow
```

Figure 10.3
An object/class instance Inspector window

Most Turbo Debugger data record Inspector windows have two panes: a top pane summarizing the record's field names/members and their current values, and a bottom pane displaying the type of the field or member highlighted in the top pane. An object/class instance Inspector window provides both of those panes, and also a third pane between them. This third pane summarizes the instance's methods or member functions, with the code address of each. (The code address takes into account polymorphic objects and the VMT.)

The object/class instance Inspector window local menus

Each of the top two panes of the object/class 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 TDINST.



```
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 top pane, which summarizes the data fields or members for the selected item, are described here.

Range...

This command is unchanged from earlier versions. It 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 or member. This command is also unchanged from earlier versions of Turbo Debugger.

Methods

This command is a *Yes/No* toggle, with *Yes* as the default condition. When it is set to *Yes*, methods or member functions are summarized in the middle pane. When it is set to *No*, the middle pane does not appear. This toggle is remembered by the next Inspector window to be opened.

Show Inherited — This command is also a *Yes/No* toggle. When it is set to *Yes*, all data fields or members and all methods or member functions are shown, whether they are defined within the type being inspected or inherited from an ancestor type. When it is set to *No*, only those fields and methods defined within the type being inspected are displayed.

Inspect — As with earlier versions of Turbo Debugger, choosing this command opens an Inspector window on the highlighted field or member. Pressing *Enter* over a highlighted field or member does the same thing.

Descend — This command has not changed from earlier versions of Turbo Debugger. The highlighted item takes the place of the item in the current Inspector window. No new Inspector window is opened. However, you cannot return to the previously inspected field, as you could if you had used the **I**nspect option.

Use **D***escend to inspect a complex data structure when you don't want to open a separate Inspector window for each item.*

New Expression... — No change from earlier versions. This command prompts you for a new data item 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 (Byte, Word, Int, Char pointer) for the item being inspected. This 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.

Hierarchy — When you choose this command, an Hierarchy window opens. For a full description of this window, see page 155.

The middle and bottom panes

The middle pane summarizes the methods of an object or the member functions of a class. The only difference between the Object Method pane's local menu and the local menu for the top pane is the absence of the **C**hange command. Unlike data fields and members, methods and member functions 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

Assembler-level debugging

This chapter is for programmers who are familiar with programming the 80x86 processor family in assembler.

You don't need to use the information in this chapter to debug your programs—but there are certain problems that may be easier to find using techniques discussed in this chapter.

We explain when you might want to use assembler-level debugging and describe the CPU window with its built-in disassembler and assembler. You then learn how to examine and modify raw hex data bytes, how to peruse the function calling stack, how to examine and modify the CPU registers, and finally how to examine and modify the CPU flags.

When source debugging isn't enough

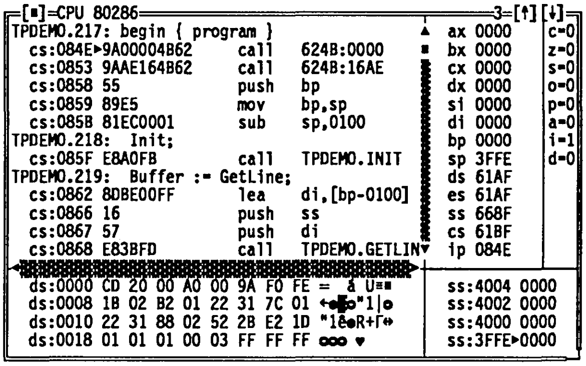
When you are debugging a program, most of the time you refer to data and code at the source level; you refer to symbol names exactly as you typed them in your source code, and you proceed through your program by executing pieces of source code.

Sometimes, however, you can gain insight into a problem by looking at the exact instructions that the compiler generated, the contents of the CPU registers, and the contents of the stack. To do this, you need to be familiar with both the 80x86 family of processors and with how the compiler turns your source code into machine instructions. 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.

C and Pascal, for example, let you write lines of source code that perform many actions at once, and Turbo Debugger lets you step one source line at a time, not one expression at a time. However, you sometimes want to know the result of executing a small piece of one source line. By stepping through your program one machine instruction at a time, you can examine intermediate results, although it does require some effort to figure out how the compiler translated your source statements into machine 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. You can 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. You can also access the underlying bytes of any data structure, display them in a number of formats, and change them.



```
[■]=CPU 80286                                         3=[↑][↓]
TPDEMO.217: begin { program }                    ▲  ax 0000   c=0
  cs:084E►9A00004B62     call   624B:0000        ■  bx 0000   z=0
  cs:0853 9AAE164B62     call   624B:16AE           cx 0000   s=0
  cs:0858 55             push   bp                  dx 0000   o=0
  cs:0859 89E5           mov    bp,sp               si 0000   p=0
  cs:085B 81EC0001       sub    sp,0100             di 0000   a=0
TPDEMO.218:  Init;                                  bp 0000   i=1
  cs:085F E8A0FB         call   TPDEMO.INIT         sp 3FFE   d=0
TPDEMO.219:  Buffer := GetLine;                     ds 61AF
  cs:0862 8DBE00FF       lea    di,[bp-0100]        es 61AF
  cs:0866 16             push   ss                  ss 668F
  cs:0867 57             push   di                  cs 61BF
  cs:0868 E83BFD         call   TPDEMO.GETLIN▼      ip 084E

  ds:0000 CD 20 00 A0 00 9A F0 FE =  å U≡■          ss:4004 0000
  ds:0008 1B 02 B2 01 22 31 7C 01 ←●█o"1|o          ss:4002 0000
  ds:0010 22 31 88 02 52 2B E2 1D "1ê●R+Γ↔          ss:4000 0000
  ds:0018 01 01 01 00 03 FF FF FF ☺☺☺ ♥             ss:3FFE►0000
```

Figure 11.1
The CPU window

Open a CPU window by choosing **V**iew | **C**PU from the menu bar. Depending on what you are viewing in the current window, the new CPU window comes up positioned at the appropriate code, data, or stack location. This provides 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 will be positioned when you choose the **C**PU command:

Current window	CPU window pane	Position
Stack window	Stack	Current SS:SP
Module window	Code	Current CS:IP
Variable window	Data*	Address of item
Inspector window	Data	Address of item
Breakpoint (if not global)	Code	Breakpoint address

*Code pane, if item in window is a routine

CPU windows have five 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 right pane is the Flags pane, showing the state of the eight CPU flags. The bottom left pane (Data 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.

In the Code pane, an arrow (▶) shows the current program location (CS:IP). In the Stack pane, an arrow (▶) shows the current stack pointer (SS:SP).

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 lets you see both where an instruction operand points in memory and the value that is about to be read or written over.

The Flags pane shows the value of each of the CPU flags.

As with all windows and panes, pressing *Alt-F10* pops up the Code pane local menu or, if control-key shortcuts are enabled, the *Ctrl* key with the first letter of the desired command gets you to it.

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. This is easier than using the **G**oto command if you just want to adjust the display slightly.

The Code pane

This pane shows the disassembled instructions at an address that you choose.

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.

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 **M**ixed 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 pound sign (#) or a period (.), followed by the static symbol name. Line numbers appear as the module name, followed by a pound sign (#) or 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.

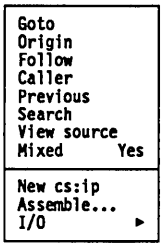
Turbo Debugger can detect an 8087, 80287, or 80387 numeric coprocessor and disassemble those instructions if a floating-point chip or emulator is present.

The instruction mnemonic **RETF** indicates that this is a far return instruction. The normal **RET** mnemonic indicates a near return.

Where possible, the target of **JMP** and **CALL** instructions is displayed symbolically. If CS:IP is a **JMP** or conditional jump instruction, an arrow (↑ or ↓) that shows jump direction will be displayed only if the executing instruction will cause the jump to occur. Also, memory addresses used by **MOV**, **ADD**, and other instructions display symbolic addresses.

The Code pane local menu

If you don't come up in the Code pane, use *Tab* or *Shift-Tab* to get there. Then press *Alt-F10* to bring up the local menu.



Goto

After choosing this command, you're prompted for the new address to go to. You can enter addresses that are outside of your program, to examine code in the BIOS ROM, inside DOS, and in resident utilities. See Chapter 9 for complete information on entering addresses.

The **P**revious command restores the Code pane to the position it had before the **G**oto command was issued.

Origin

Positions you at the current program location as indicated by the CS:IP register pair. This command is useful when you want to return to where you started.

The **P**revious command restores the Code pane to the position it had before the **O**rigin command was issued.

Follow

Positions you at the destination address of the currently highlighted instruction. The Code pane is repositioned to display the code at the address where the currently highlighted instruction will transfer control. For conditional jumps, the address is shown as if the jump occurred.

This command can be used with the **CALL, JMP**, conditional jump (**JZ, JNE, LOOP, JCXZ**, and so forth) and **INT** instructions.

The **P**revious command restores the Code pane to the position it had before the **F**ollow command was selected.

Caller Positions you at the instruction that called the current interrupt or subroutine.

This command won't always work. If the interrupt routine or subroutine has pushed data items onto the stack, sometimes Turbo Debugger can't figure out where the routine was called from.

The **P**revious command restores the Code pane to the position it had before the **C**aller command was selected.

Previous Restores the Code pane position to the address before the last command that explicitly changed the display address. Using the arrow keys and *PgUp* and *PgDn* does not cause the position to be remembered.

When you choose **P**revious, the Code pane position is remembered, so that repeated use of the **P**revious command causes the Code pane to switch back and forth between two addresses.

Search Lets you enter an instruction or byte list to search for. Enter an instruction exactly as you would with the **A**ssemble command.

Be careful which instructions you try to search for; you should only search for instructions that don't change the bytes they assemble to, depending on their location in memory. For example, searching for the following instructions is no problem:

```
PUSH DX
POP  [DI+4]
ADD  AX,100
```

but searching for the following instructions can cause unpredictable results:

```
JE   123
CALL MYFUNC
LOOP 100
```

You can also enter a byte list instead of an instruction. See Chapter 9 for more on entering byte lists.

Mixed — Toggles between the three ways of displaying disassembled instructions and source code:

No	No source code is displayed, only disassembled instructions.
Yes	Source code lines appear before the first disassembled instruction for that source line. The pane is set to this display mode if your current module is a high-level language source module.
Both	Source code lines replace disassembled lines for those lines that have corresponding source code; otherwise, the disassembled instruction appears. Use this mode when you are debugging an assembler module, and you want to see the original source code, instead of the corresponding disassembled instruction. The pane is set to this display mode if your current module is an assembler source module.

New CS:IP — Sets the program location counter (CS:IP registers) to the currently highlighted address. When you rerun your program, execution starts at this address. This is useful when you want to skip over a piece of code without executing it.

☞ *Use this command with extreme care.* If you adjust the CS:IP to a location where the stack is in a different state than at the current CS:IP, you will almost certainly crash your program. Do not use this command to set the CS:IP to an address outside of the current routine.

Assemble... — Assembles an instruction, replacing the one at the currently highlighted location. You are prompted for the instruction to assemble. See the section called "The assembler" in this chapter (page 180) for more details.

You can also invoke this command by simply starting to type the statement you want to assemble. When you do this, a dialog box appears exactly as if you had specified **A**ssemble.

I/O Reads or writes a value in the CPU's I/O space and lets you examine the contents of I/O registers on cards and write things to them.

```
In byte
Out byte
Read word
Write word
```

It pops up this menu.

In Byte

Reads a byte from an I/O port. You are prompted for the I/O port whose value you want to examine. Use the **R**ead Word option to read from a word-sized I/O port.

Out Byte

Writes a byte to an I/O port. You are prompted for the I/O port to write to and the value you want to write. Use the **W**rite Word option to write to a word-sized I/O port.

Read Word

Reads a word from an I/O port. You are prompted for the I/O port whose value you want to examine. Use the **I**n Byte option to read from a byte-sized I/O port.

Write Word

Writes a word to an I/O port. You are prompted for the I/O port to write to and the value you want to write. Use the **O**ut Byte option to write to a byte-sized I/O port.

IN and **OUT** instructions access the I/O space where peripheral device controllers (such as serial cards, disk controllers, and video adapters) reside.

Be careful when you use these commands. Some I/O devices consider reading their ports to be a significant event that causes the device to perform some action, such as resetting status bits or loading a new data byte into the port. You may disrupt the normal operation of the program you are debugging or the device with indiscriminate use of these commands.

The Register and Flags panes

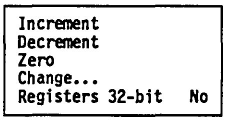
The Register pane, which is the top pane to the right of the Code pane, shows the contents of the CPU registers.

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

The Register pane local menu

Press *Alt-F10* to pop up the Register pane local menu. Or, if control-key shortcuts are enabled, use the *Ctrl* key with the first letter of the desired command to access the command.



```
Increment
Decrement
Zero
Change...
Registers 32-bit   No
```

Increment Adds 1 to the value in the currently highlighted register. This is an easy way to make small adjustments in the value of a register to compensate for "off-by-one" bugs.

Decrement Subtracts 1 from the value in the currently highlighted register.

Zero Sets the value of the currently highlighted register to 0.

Change... Changes the value of the currently highlighted register. You are prompted for the new value. You can make full use of the expression evaluator to enter a new value.

You can also invoke this command by simply starting to type the new value for the register. A dialog box appears exactly as if you had specified the **C**hange command.

Registers 32-bit

On an 80386 processor, toggles between displaying the CPU registers as 16-bit or 32-bit values. You will usually see 16-bit registers, unless you use this command to set the display to 32-bit registers. You really need to see 32-bit registers only if you're debugging a program that uses the 32-bit addressing capabilities of the 386 chip. If you are debugging an ordinary program that uses only normal 16-bit addressing, use the 16-bit register display.

The Flags pane local menu

Toggle

Press *Alt-F10* to pop up the Flags pane local menu or, if control-key shortcuts are enabled, use the *Ctrl* key with the first letter of the desired command to access the command.

Toggle

Sets the value of the flag to 0 if it was 1, and to 1 if it was 0. The value 0 corresponds to "clear," and 1 indicates "set." You can also press *Enter* to toggle the value of the currently highlighted flag.

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 the DS register name if the segment value is the same as the current DS register.

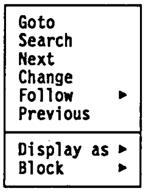
Next, the raw display of one or more data items is displayed. The format of this area depends on the display format selected with the **D**isplay As local menu command. If you choose one of the floating-point display formats (**C**omp, **F**loat, **R**eal, **D**ouble, **E**xtended), a single floating-point number is displayed on each line. **B**yte format displays 8 bytes per line, **W**ord format displays 4 words per line, and **L**ong 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. Turbo Debugger 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.

If you use the Data pane to examine the contents of the display memory, the ROM BIOS data area, or the vectors in low memory, you will see the values that are there when the program being debugged runs, *not* the actual values in memory when Turbo Debugger is running. These are not the same values that are in these memory areas at the time you look at them. Turbo Debugger detects when you're accessing areas of memory that it uses as well, and it gets the correct data value from where it stores the user program's copy of these data areas.

The Data pane local menu

Once you are positioned in the Data pane, press *Alt-F10* to pop up the local menu or, if control-key shortcuts are enabled, use the *Ctrl* key with the first letter of the desired command to access it.



Goto

Positions you at an address in your data. Enter the new address you want to go to. You can enter addresses inside DOS, in resident utilities, or outside of your program, which lets you examine data in the BIOS data area. See Chapter 9 for a complete discussion of how to enter addresses.

Search

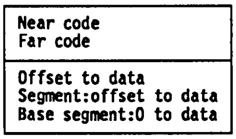
Searches for a character string, starting at the current memory address as indicated by the cursor position. Enter the byte list to search for. The search does not wrap around from the end of the segment to the beginning. See Chapter 9 for a complete discussion of byte lists.

Next Searches for the next instance of the byte list you previously specified with the **S**earch command.

Change... Lets you change the bytes at the current cursor location. If you're over an ASCII display or the format is **B**yte, you're prompted for a byte list. Otherwise, you're prompted for an item of the current display type. See Chapter 9 for a discussion of byte lists.

You can also invoke this command by simply starting to type the new value or values. This brings up a dialog box exactly as if you had chosen the **C**hange command.

Follow This command opens a menu that lets you follow near or far pointer chains.



Near Code

This command interprets the word under the cursor in the Data pane as an offset into the current code segment as specified by the CS register. The Code pane becomes the current pane and is positioned to this address.

Far Code

This command interprets the doubleword under the cursor in the Data pane as a far address (segment and offset). The Code pane becomes the current pane and is positioned to this address.

Offset to Data

This command lets you follow word (near, offset only) pointer chains. The Data pane is set to the offset specified by the word in memory at the current cursor location.

Segment:Offset to Data

This command lets you follow long (far, segment, and offset) pointer chains. The Data pane is set to the offset specified by the two words in memory at the current cursor location.

Base Segment:0 to Data

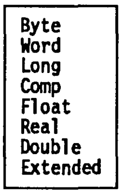
This command interprets the word under the cursor as a segment address and positions the Data pane to the start of that segment.

Previous

Restores the Data pane address to the address before the last command that explicitly changed the display address. Using the arrow keys and *PgUp* and *PgDn* does not cause the position to be remembered.

Turbo Debugger maintains a stack of the last five addresses, so you can backtrack through multiple uses of the **F**ollow menu or **G**oto commands.

Display As



Lets you choose how data appears in the Data pane. You can choose from any data format used by C, Pascal, and assembler. The menu options are described here.

Byte

Sets the Data pane to display as hexadecimal bytes. This corresponds to the C **char** data type, the Pascal double type, and the Pascal Byte type.

Word

Sets the Data pane to display as word hexadecimal numbers. The 2-byte hexadecimal value is shown. This corresponds to the C **int** data type and the Pascal Word type.

Long

Sets the Data pane to display as long hexadecimal integers. The 4-byte hex value is shown. This corresponds to the C **long** data type and the Pascal Longint type.

Comp

Sets the Data pane to display 8-byte integers. The decimal value of the integer is shown. This is the Pascal Comp (IEEE) data type.

Float

Sets the Data pane to display as short floating-point numbers. The scientific notation floating-point value is shown. This is the same as the C **float** data type and the Pascal Single (IEEE) type.

Real

Sets the Data pane to display Pascal's 6-byte floating-point numbers. The scientific notation floating-point value is shown. This is the Pascal Real type.

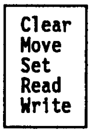
Double

Sets the data pane to display 8-byte floating-point numbers. The scientific notation floating-point value is shown. This is the same as the C **long double** data type, the Pascal Double type, and the assembler **TBYTE** type.

Extended

Sets the Data pane to display 10-byte floating-point numbers. The scientific notation floating-point value is shown. This is the internal format used by the 80x87 coprocessor. It also corresponds to the C **long double** data type and the Pascal Extended (IEEE) type.

Block



Lets you manipulate blocks of memory. You can move, clear and set memory blocks, and read and write memory blocks to and from disk files. **Block** brings up the pop-up menu shown.

Clear

Sets a contiguous block of memory to zero (0). You are prompted for the address and the number of bytes to clear.

Move

Copies a block of memory from one address to another. You are prompted for the source address, the destination address, and how many bytes to copy.

Set

Sets a contiguous block of memory to a specific byte value. You are prompted for the address of the block, how many bytes to set, and the value to set them to.

Read

Reads all or a portion of a file into a block of memory. You are prompted first for the file name to read from, then for the address to read it into, and finally for how many bytes to read.

Write

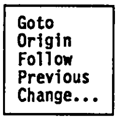
Writes a block of memory to a file. You are prompted first for the file name to write to, then for the address of the block to write and how many bytes to write.

The Stack pane

The Stack pane, in the lower right corner of the CPU window, shows the contents of the stack.

The Stack pane local menu

At the Stack pane, press *Alt-F10* to pop up the local menu or, if control-key shortcuts are enabled, use the *Ctrl* key with the first letter of the desired command to access the command.



Goto Positions you at an address in the stack. Enter the new stack address. If you want, you can enter addresses outside your program's stack, although you would usually use the Data pane to examine arbitrary data outside your program. See Chapter 9 for information about how to enter addresses.

The **P**revious command restores the Stack pane to the position it had before the **G**oto command was issued.

Origin Positions you at the current stack location as indicated by the SS:SP register pair. This command is useful when you want to return to where you started.

The **P**revious command restores the Stack pane to the position it had before the **O**rigin command was issued.

Follow Positions you at the word in the stack pointed to by the currently highlighted word. This is useful for following stack-frame threads back to a calling function.

The **P**revious command restores the Stack pane to the position it had before the **F**ollow command was issued.

Previous Restores the Stack pane position to the address before the last command that explicitly changed the display address. Using the arrow keys and *PgUp* and *PgDn* does not cause the position to be remembered.

Repeated use of the **P**revious command causes the Stack pane to switch back and forth between two addresses.

Change Lets you enter a new word value for the currently highlighted stack word.

You can also invoke this command by simply starting to type the new value for the highlighted stack item. A dialog box will appear, exactly as if you had specified the **C**hange command.

The assembler

Via the **A**ssemble command in the Code pane local menu, Turbo Debugger lets you assemble instructions for the 8086, 80186, 80286, 80386, and 80486 processors, and also for the 8087, 80287, and 80387 numeric coprocessors.

When you use Turbo Debugger's built-in assembler to modify your program, the changes you make are not permanent. If you reload your program using the **R**un | **P**rogram Reset command, or if you load another program using the **F**ile | **O**pen command, you'll lose any changes you've made.

Normally you use the assembler to test an idea for fixing your program. Once you've verified that the change works, you must change your source code and recompile and link your program.

The following sections describes the differences between the built-in assembler and the syntax accepted by Turbo Assembler.

Operand address size overrides

For the call (**CALL**), jump (**JMP**), and conditional jump (**JNE**, **JL**, and so forth) instructions, the assembler automatically generates the smallest instruction that can reach the destination address. You can use the NEAR and FAR overrides before the destination address to assemble the instruction with a specific size. For example,

```
CALL FAR XYZ
JMP  NEAR A1
```

Memory and immediate operands

When you use a symbol from your program as an instruction operand, you must tell the built-in assembler whether you mean the contents of the symbol or the address of the symbol. If you use just the symbol name, the assembler treats it as an address, exactly as if you had used the assembler **OFFSET** operator before it. If you put the symbol inside brackets ([ ]), it becomes a memory reference. For example, if your program contains the data definition

```
A    DW 4
```

then [A] references the area of memory where *A* is stored.

When you assemble an instruction or evaluate an assembler expression to refer to the contents of a variable, use the name of the variable alone or between brackets:

```
mov dx,a
mov ax,[a]
```

To refer to the address of the variable, use the **OFFSET** operator:

```
mov ax,offset a
```

Operand data size overrides

For some instructions, you must specify the operand size using one of the following expressions before the operand:

```
BYTE PTR
WORD PTR
```

Here are examples of instructions using these overrides:

```
add BYTE PTR[si],10
mov WORD PTR[bp+10],99
```

In addition to these size overrides, you can use the following overrides to assemble 8087/80287/80387 numeric coprocessor instructions:

```
DWORD PTR
QWORD PTR
TBYTE PTR
```

Here are some examples using these overrides:

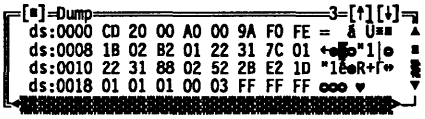
```
fild QWORD PTR[bx]
stp  TBYTE PTR[bp+4]
```

String instructions

When you assemble a string instruction, you must include the size (byte or word) as part of the instruction mnemonic. The assembler does not accept the form of the string instructions that uses a sizeless mnemonic with an operand that specifies the size. For example, use **STOSW** rather than **STOS WORD PTR**[di].

The Dump window

The Dump window shows you a raw data dump of any area of memory. It works exactly like the Data pane in the CPU window.



```
[■]=Dump================================3=[↑][↓]
 ds:0000 CD 20 00 A0 00 9A F0 FE =  å U
 ds:0008 1B 02 B2 01 22 31 7C 01 ←  "1|
 ds:0010 22 31 88 02 52 2B E2 1D "1 R+Γ↔
 ds:0018 01 01 01 00 03 FF FF FF    ♥
```

Figure 11.2
The Dump window

See "The Data pane local menu" section earlier in this chapter (page 175) for a description of the contents and local menu for this window.

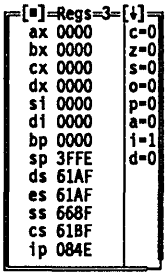
Typically, you'd use this window when you're debugging an assembler program at the source level, and you want to take a low-level look at some data areas. Use **View | Dump** to open a Dump window.

You can also 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. Use **View | Dump** to get a Dump window that's positioned to the data in the Inspector window.

The Registers window

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.

Figure 11.3
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
```

You can shrink the size of your Module window and put up a Registers window alongside it.

See "The Register pane local menu" (page 173) and "The Flags pane local menu" (page 174) sections earlier in this chapter for a description of the contents and local menus for this window.

Use this window when you're debugging an assembler program at the source level and want to look at the register values.

Turbo C code generation

The Turbo C compiler does a number of predictable things when it generates machine code. Once you become familiar with the compiler, you'll quickly see exactly how the machine instructions correspond to your source code.

Function return values are placed in the following registers:

Return type	Register(s)
int	AX
long	DX:AX
float	ST(0)
double	ST(0)
long double	ST(0)
near *	AX
far *	DX:AX

The compiler places heavily used **int** and **near** pointers into registers, using first the SI register, then the DI register.

Your autovariables and function-calling parameters are accessed from SS:BP.

The AX, BX, CX, and DX registers are not necessarily preserved across function calls.

Registers are always used as word registers, not as byte registers, even if you use **char** data types.

Switch statements can be compiled into one of three forms, depending on which will produce the most efficient code:

- conditional jumps as if the switch were an **if...else** chain
- a jump table of code addresses
- a jump table of switch values and code addresses

Refer to your Turbo C manuals for more information on Turbo C code generation.

C H A P T E R

12

The 80x87 coprocessor chip and emulator

This chapter is for programmers who are familiar with the operation of the 80x87 math coprocessor.

If your program uses floating-point numbers, Turbo Debugger lets you examine and change the state of the numeric coprocessor or software emulator. You don't need to use the capabilities described in this chapter to debug programs that use floating-point numbers, although some very subtle bugs may be easier to find.

In this chapter, we discuss the differences between the 80x87 chip and the software emulator. We also describe the Numeric Processor window and show you how to examine and modify the floating-point registers, the status bits, and the control bits.

The 80x87 chip vs. the emulator

Turbo Debugger automatically detects whether your program is using the math chip or the emulator and adjusts its behavior accordingly.

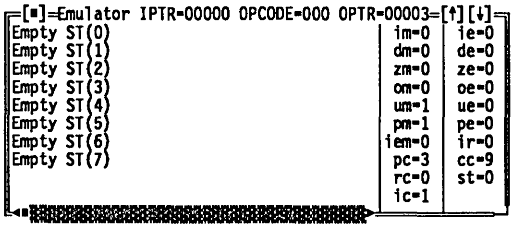
Note that most programs use either the emulator or the math chip, not both within the same program. If you have written special assembler code that uses both, Turbo Debugger won't be able to show you the status of the math chip; it will report on the emulator only.

The Numeric Processor window

You create a Numeric Processor window by choosing the **V**iew | **N**umeric Processor command from the menu bar. The line at the top of the window shows the current instruction pointer, data pointer, and instruction opcode. The data pointer and instructions pointer are both shown as 20-bit physical addresses. You can convert these addresses to a segment and offset form by using the first four digits as the segment value, and the last digit as the offset value.

For example, if the top line shows `IPTR=5A669`, you can treat this as the address 5a66:9 if you want to examine the current data and instruction in a CPU window. This window has three panes: The left pane (Register pane) shows the contents of the floating-point registers, the middle pane (Control pane) shows the control flags, and the right pane (Status pane) shows the status flags.

Figure 12.1
The Numeric Processor window



```
[■]=Emulator IPTR=00000 OPCODE=000 OPTR=00003=[↑][↓]
Empty ST(0)                                 im=0    ie=0
Empty ST(1)                                 dm=0    de=0
Empty ST(2)                                 zm=0    ze=0
Empty ST(3)                                 om=0    oe=0
Empty ST(4)                                 um=1    ue=0
Empty ST(5)                                 pm=1    pe=0
Empty ST(6)                                iem=0    ir=0
Empty ST(7)                                 pc=3    cc=9
                                            rc=0    st=0
                                            ic=1
```

The top line shows you information about the last floating-point operation that was executed. The IPTR shows the 20-bit physical address from which the last floating-point instruction was fetched. The OPCODE shows the instruction type that was fetched. The OPTR shows the 20-bit physical address of the memory address that the instruction referenced, if any.

The Register pane

The 80-bit floating-point registers

The Register pane shows each of the floating-point registers, ST(0) to ST(7), along with its status (valid/zero/special/empty). The contents are shown as an 80-bit floating-point number.

If you've zoomed the Numeric Processor window (by pressing *F5*) or made it wider by using **W**indow | **S**ize/Move, you'll also see the floating-point registers displayed as raw hex bytes.

The Register pane local menu

```
Zero
Empty
Change
```

To bring up the Register pane local menu, press *Alt-F10,* or use the *Ctrl* key with the first letter of the desired command to directly access the command.

Zero

Sets the value of the currently highlighted register to zero.

Empty

Sets the value of the currently highlighted register to empty. This is a special status that indicates that the register no longer contains valid data.

Change

Loads a new value into the currently highlighted register. You are prompted for the value to load. You can enter an integer or floating-point value, using the current language's expression parser. The value you enter will be automatically converted to the 80-bit temporary real format used by the numeric coprocessor.

You can also invoke this command by simply starting to type the new value for the floating-point register. A dialog box will appear exactly as if you had specified the **C**hange command.

The Control pane

The control bits

The following table lists the different control flags and how they appear in the Control pane:

Name in pane	Flag description
im	Invalid operation mask
dm	Denormalized operand mask
zm	Zero divide mask
om	Overflow mask
um	Underflow mask
pm	Precision mask
iem	Interrupt enable mask (8087 only)
pc	Precision control
rc	Rounding control
ic	Infinity control

The Control pane local menu

```
Toggle
```

Press *Tab* to go to the Control pane, then press *Alt-F10* to pop up the local menu. (Alternatively, you can use the *Ctrl* key with the first letter of the desired command to access it.)

Toggle

Cycles through the values that the currently highlighted control flag can be set to. Most flags can only be set or cleared (0 or 1), so this command just toggles the flag to the other value. Some other flags have more than two values; for those flags, this command increments the flag value until the maximum value is reached, and then it sets it back to zero.

You can also toggle the control flag values by highlighting them and pressing *Enter.*

The Status pane

The status bits

The following table lists the different status flags and how they appear in the Status pane:

Name in pane	Flag description
ie	Invalid operation
de	Denormalized operand
ze	Zero divide
oe	Overflow
ue	Underflow
pe	Precision
ir	Interrupt request
cc	Condition code
st	Stack top pointer

The Status pane local menu

```
Toggle
```

Press *Tab* to move to the Status pane, then press *Alt-F10* to pop up the local menu. (You can also use the *Ctrl* key with the first letter of the desired command to access the command directly.)

Toggle

Cycles through the values that the currently highlighted status flag can be set to. Most flags can only be set or cleared (0 or 1), so this command just toggles the flag to the other value. Some other flags have more than two values; for those flags, this command increments the flag value until the maximum value is reached, and then sets it back to zero.

You can also toggle the status flag values by highlighting them and pressing *Enter*.

C H A P T E R

13

Command reference

Now that you've read about all the commands, here's a quick summary. This chapter lists and describes

- all the single-keystroke commands available on the function and other keys
- all the menu bar commands and the commands for the local menu of each window type
- keystrokes used in the two types of panes (those in which you enter text and those from which you select an item)
- keystrokes for moving and resizing windows

Hot keys

A hot key is a key that performs its action no matter where you are in the Turbo Debugger environment. Table 13.1 on page 192 lists all the hot keys.

Table 13.1
The function key and hot key commands

Key	Menu command	Function
F1		Brings up context-sensitive help
F2	**B**reakpoints | **T**oggle	Sets breakpoint at cursor position
F3	**V**iew | **M**odule	Module pick list
F4	**R**un | **G**o to Cursor	Runs to cursor position
F5	**W**indow | **Z**oom	Zooms/unzooms current window
F6	**W**indow | **N**ext Window	Goes to next window
F7	**R**un | **T**race Into	Executes single source line or instruction
F8	**R**un | **S**tep Over	Executes single source line or instruction, skipping calls
F9	**R**un | **R**un	Runs program
F10		Invokes the menu bar, takes you out of menus
Alt-F1	**H**elp | **P**revious Topic	Brings up last help screen
Alt-F2	**B**reakpoints | **A**t	Sets breakpoint at an address
Alt-F3	**W**indow | **C**lose	Closes current window
Alt-F4	**R**un | **B**ack Trace	Reverses program execution
Alt-F5	**W**indow | **U**ser Screen	Shows your program's screen
Alt-F6	**W**indow | **U**ndo Close	Reopens the last-closed window
Alt-F7	**R**un | **I**nstruction Trace	Executes a single instruction
Alt-F8	**R**un | **U**ntil Return	Runs until return from function
Alt-F9	**R**un | **E**xecute To	Runs to a specified address
Alt-F10		Invokes the window's local menu
Alt-1—9		Switch to numbered window 1 through 9
Alt-Space		Goes to the System menu
Alt-B		Goes to the **B**reakpoints menu
Alt-D		Goes to the **D**ata menu
Alt-F		Goes to the **F**ile menu
Alt-H		Goes to the **H**elp menu
Alt-O		Goes to the **O**ptions menu
Alt-R		Goes to the **R**un menu
Alt-V		Goes to the **V**iew menu
Alt-W		Goes to the **W**indow menu
Alt-X	**F**ile | **Q**uit	Quits Turbo Debugger and returns you to DOS
Alt=	**O**ptions | **M**acros | **C**reate	Defines a keystroke macro
Alt-	**O**ptions | **M**acros | **S**top Recording	Ends a macro recording

Table 13.1: The function key and hot key commands (continued)

Key	Menu command	Function
Ctrl-F2	**Run** | **Program Reset**	Stops debug session and resets the program to start again
Ctrl-F4	**Data** | **Evaluate**	Evaluates an expression
Ctrl-F5	**Window** | **Size/Move**	Initiates window moving or resizing
Ctrl-F7	**Data** | **Add Watch**	Adds a variable to the Watches window
Ctrl-F8	**Breakpoints** | **Toggle**	Toggles a breakpoint at cursor
Ctrl-F9	**Run** | **Run**	Runs a program
Ctrl-F10		Invokes the window's local menu
Ctrl →		Shifts the starting address in a Code, Data, or Stack pane in a CPU window 1 byte up
Ctrl ←		Shifts the starting address in a Code, Data, or Stack pane in a CPU window 1 byte down
Ctrl-A		Moves to previous word
Ctrl-C		Scrolls down one screen
Ctrl-D		Moves right one column
Ctrl-E		Moves up one line
Ctrl-F		Moves to next word
Ctrl-R		Scrolls up one screen
Ctrl-S		Moves left one column
Ctrl-X		Moves down one line
Shift-F1	**Help** | **Index**	Goes to the index for online help
Shift-Tab		Moves cursor to previous window pane or dialog box item
Shift → *Shift ←* *Shift ↑* *Shift ↓*		Moves cursor between the panes in a window (the pane in the direction of the arrow becomes the active pane.)
Esc		Closes an Inspector window, goes out of menus
Ins		Starts text block selection (highlight); use ← and → to highlight
Tab	**Window** | **Next Pane**	Moves cursor to next window pane or dialog box item

Commands from the menu bar

You invoke the menu bar by pressing the *F10* key; you can then go directly to one of the individual menus by

- cursoring to the menu title and pressing *Enter*
- pressing the highlighted letter of the menu title

You can also open a menu directly (without first moving to the menu bar) by pressing *Alt* in combination with the first letter of the menu name you desire.

The ≡ (System) menu

Restore Standard	Restores standard window layout
Repaint Desktop	Redisplays entire screen
About	Displays information about Turbo Debugger

The File menu

Open	Opens a new program to debug
Change Dir	Changes to new disk or directory
Get Info	Displays program information
DOS Shell	Starts a DOS command processor
Resident	Causes Turbo Debugger to terminate and stay resident
Symbol Load	Loads symbol table independent of .EXE file
Table Relocate	Sets base segment of symbol table
Quit	Returns to DOS

The View menu

Breakpoints	Displays breakpoints
Stack	Displays function-calling stack
Log	Displays log of events and data
Watches	Displays variables being watched
Variables	Displays global and local variables
Module	Displays program source module
File	Displays disk file as ASCII or hex
CPU	Displays CPU instructions, data, stack
Dump	Displays raw data dump

Registers	Displays CPU registers and flags
Numeric Processor	Displays coprocessor or emulator
Execution History	Displays assembler code saved for backtracking or keystroke playback
OOP Hierarchy	Displays object or class type list and hierarchy tree
Another	
Module	Makes another Module window
Dump	Makes another Dump window
File	Makes another File window

The Run menu

Run	Runs your program without stopping
Go To Cursor	Runs to current cursor location
Trace Into	Executes one source line or instruction
Step Over	Traces, skipping calls
Execute To	Runs to specified address
Until Return	Runs until function returns
Animate	Continuously steps your program
Back Trace	Reverses program execution for one source line or instruction
Instruction Trace	Executes a single instruction
Arguments	Sets program command-line arguments
Program Reset	Reloads current program

The Breakpoints menu

Toggle	Toggles breakpoint at cursor
At	Sets breakpoint at specified address
Changed Memory Global	Sets global breakpoint on memory area
Expression True Global	Sets global breakpoint on expression
Delete All	Removes all breakpoints

The Data menu

Inspect	Inspects a data object
Evaluate/Modify	Evaluates an expression
Add Watch	Adds variable to Watches window
Function Return	Inspects current routine's return value

The Options menu

Language	Sets expression language from source module
Macros	
Create	Defines a keystroke macro
Stop Recording	Ends the recording session
Remove	Removes a keystroke macro
Delete All	Removes all keystroke macros
Display Options	Lets you set screen display options (screen swapping, size, tabs)
Path for Source	Directory list for source files
Save Options	Saves options, screen layout, and macros to disk
Restore Options	Restores options from disk

The Window menu

Zoom	Zooms window to full screen size and back
Next	Activates successive windows open onscreen
Ne**x**t Pane	Goes to the next pane in a window
Size/Move	Moves window or changes its size
Iconize/Restore	Reduces window to a small symbol or restores it
Close	Closes window
Undo Close	Reopens the last window closed
Dump Pane to Log	Writes current pane to Log window
User Screen	Displays your program output
Open window list	Displays list of open windows to activate
Window Pick	Displays a menu of open menus, if more than 9 are open onscreen

The Help Menu

Index	Goes to the index for online help
Previous Topic	Brings up last help screen
Help on Help	Accesses online help on the help system

The local menu commands

Each type of window and each pane within a window has a different local menu.

You invoke the local menu for the current window by pressing *Alt-F10*. If control-key shortcuts are enabled, you can go directly to one of the individual menu items by pressing the *Ctrl* key in combination with the first letter of the item you desire. (Use the installation program TDINST to enable control-key shortcuts, if they've been disabled.)

The menus in this section are arranged in alphabetical order to make lookups easier.

The following sections describe the local menu for each window and pane.

Some panes have shortcuts to commonly used commands on their local menu. In the following section, these special keys are listed before the menu commands for the pane to which they apply. In many panes, the *Enter* key is a shortcut to examining or changing the currently highlighted item. The *Del* key often invokes the local menu command that deletes the highlighted item. Some panes let you start typing letters or numbers without first invoking a local menu command. In these cases, the dialog box for one of the local menu items pops up to accept your input.

Breakpoints window

The Breakpoints window has two panes, the List pane on the left and the Detail pane on the right. Only the List pane has a local menu.

Set Options	Sets breakpoint actions, conditions, pass count, and enable/disable
Hardware Options	Lets you set hardware breakpoints
Add	Adds a new breakpoint
Remove	Removes highlighted breakpoint
Delete All	Deletes all breakpoints
Inspect	Looks at code where breakpoint is set

Del is the shortcut for **R**emove in this window.

The CPU window menus

The CPU window has five panes, each with a local menu: the Code pane, the Data pane, the Stack pane, the Register pane, and the Flags pane.

Code pane	**G**oto	Displays code at new address
	Origin	Displays code at CS:IP
	Follow	Displays code at **JMP** or **CALL** target
	Caller	Displays code at calling function
	Previous	Displays code at last address
	Search	Searches for instruction or bytes
	View Source	Switches to Module window
	Mixed	Mixes source code with disassembly: *No/Yes/Both*
	New CS:IP	Sets CS:IP to execute at new address
	Assemble	Assembles instruction at cursor **I**/O
	In Byte	Reads a byte from an I/O location
	Out Byte	Writes a byte to an I/O location
	Read Word	Reads a word from an I/O location
	Write Word	Writes a word to an I/O location

Typing any character is a shortcut for the Assemble local menu command in this pane.

Data pane	**G**oto	Displays data at new address
	Search	Searches for string or data bytes
	Next	Searches again for next occurrence
	Change	Changes data bytes at cursor address
	Follow	
	Near Code	Sets Code pane to the near address under the cursor
	Far Code	Sets Code pane to the far address under the cursor
	Offset to Data	Sets Data pane to the near address under the cursor
	Segment:Offset to Data	Sets Data pane to the far address under the cursor
	Base Segment:0 to Data	Sets Data pane to start of segment that contains the address under the cursor
	Previous	Displays data at last address
	Display As	
	Byte	Displays hex bytes
	Word	Displays hex words
	Long	Displays hex 32-bit long words

	Comp	Displays 8-byte Pascal comp integers
	Float	Displays short (4-byte) floating-point numbers (Pascal singles, C floats)
	Real	Displays 6-byte floating-point numbers (Pascal reals)
	Double	Displays 8-byte floating-point numbers (Pascal and C doubles)
	Extended	Displays 10-byte floating-point numbers (C long double, Pascal extended)
Block		
	Clear	Sets memory block to zero
	Move	Moves memory block
	Set	Sets memory block to value
	Read	Reads from file to memory
	Write	Writes from memory to file

Typing any character is a shortcut for the **C**hange local menu command in this pane.

Flags pane

Toggle	Sets or clears highlighted flag

Pressing *Enter* or *Spacebar* is a shortcut for the **T**oggle local menu command in this pane.

Register pane

Increment	Adds one to highlighted register
Decrement	Subtracts one from highlighted register
Zero	Clears highlighted register
Change	Sets highlighted register to new value
Registers 32-bit	Toggles 32-bit register display: *No/ Yes*

Typing any character is a shortcut for the **C**hange local menu command in this pane.

Stack pane

Goto	Displays stack at new address
Origin	Displays data at SS:SP
Follow	Displays code pointed to by current item
Previous	Restores display to last address
Change	Allows you to edit information

Typing any character is a shortcut for the **C**hange local menu command in this pane.

Dump window

The Dump window is identical to the Data pane of the CPU window. Its local menu is identical to the Data pane local menu.

File window

The File window shows the contents of the disk file as hex bytes or as a disk file.

Goto	Displays line number or hex offset
Search	Searches for string or data bytes
Next	Searches again for next occurrence
Display As	Sets file display mode: ASCII/Hex
File	Switches to view new file
Edit	Edits file or changes bytes at cursor

Typing any character is a shortcut for the **S**earch local menu command.

Log window menu

The Log window shows messages sent to the log.

Open Log File	Starts logging to a file
Close Log File	Stops logging to a file
Logging	Toggles logging: *No/Yes*
Add Comment	Writes user comment to log
Erase Log	Clears all log messages

Typing any character is a shortcut for the **A**dd Comment local menu command.

Module window

The Module window shows the source file for the program module.

Inspect	Shows contents of variable under cursor
Watch	Adds variable under cursor to watch list
Module	Changes to display different module

File	Changes to display different file
Previous	Displays last module and position
Line	Displays source at line in module
Search	Searches for text string
Next	Searches for next occurrence of string
Origin	Displays current program location
Goto	Shows source or instructions at address
Ed**i**t	Starts editor to edit source file

Typing any character is a shortcut for the **G**oto local menu command.

Numeric Processor window

The Numeric Processor window has three panes: the Register pane, the Status pane, and the Control pane.

Register pane

These are the local menu commands in this pane:

Zero	Clears the highlighted register
Empty	Sets the highlighted register to empty
Change	Sets the highlighted register to a value

Typing any character is a shortcut for the **C**hange local menu command in this pane.

Status pane

This is the local menu command in this pane:

Toggle	Cycles through valid flag values

Pressing *Enter* is a shortcut for the **T**oggle local menu command in this pane.

Control pane

This is the local menu command in this pane:

Toggle	Cycles through valid flag values

Pressing *Enter* is a shortcut for the **T**oggle local menu command in this pane.

Hierarchy window

OOP

The Hierarchy window has two panes, the Object Type/Class List pane and the Hierarchy Tree pane. It also has a third pane, the Parent Tree pane, if you are running a C++ program with multiple inheritance.

Object Type/Class List pane	**Inspect**	Shows contents of highlighted object or class type
	Tree	Moves to the Hierarchy Tree pane
Hierarchy Tree pane	**Inspect**	Shows contents of highlighted object or class type
	Parents	Toggles whether Parent Tree pane is displayed if you are running a C++ program with multiple inheritance
Parent Tree pane	**Inspect**	Shows contents of highlighted object or class type

Registers window menu

The Registers window is identical to the Register and Flags panes of the CPU window. Its local menus are identical to the Register pane local menu and the Flags pane local menu.

Stack window

The Stack window shows the currently active functions.

Inspect	Shows source code for highlighted function
Locals	Shows local variables for highlighted function

Pressing *Enter* is a shortcut for the **Inspect** local menu command.

Variables window

The Variables window has two panes, each with a local menu: The Global Symbol pane and the Local Symbol pane.

Global Symbol pane

Inspect	Shows contents of highlighted symbol
Change	Changes value of highlighted symbol

Pressing *Enter* is a shortcut for the **I**nspect local menu command in this pane.

Local Symbol pane

Inspect	Shows contents of highlighted symbol
Change	Changes value of highlighted symbol

Pressing *Enter* is a shortcut for the **I**nspect local menu command in this pane.

Watches window

The Watches window has a single pane that shows the names and values of the variables you're watching.

Watch	Adds a variable or expression to watch
Edit	Lets you edit a watch variable or expression
Remove	Deletes highlighted variable or expression
Delete All	Deletes all watch variables or expressions
Inspect	Shows contents of highlighted variable or expression
Change	Changes contents of highlighted variable; does not affect expressions

The following keys are shortcuts to local menu commands in this window:

any character	**W**atch
Enter	**E**dit
Del	**R**emove

Inspector window

An Inspector window shows the contents of a data item.

Range	Selects array members to inspect
Change	Changes the value of highlighted item
Inspect	Opens new Inspector window for highlighted item
Descend	Expands highlighted item into this Inspector window
New Expression	Inspects a new expression in this Inspector window
Type Cast	Type casts highlighted item to new type

Object Type/ Class Inspector window

OOP

Object type/class Inspector windows have two panes that show the contents (data fields or members, and methods or member functions) of an object or class. Their local menus, the same for both panes, are quite different from the local menu of regular Inspector windows.

Inspect	Shows the contents of the highlighted type
Hierarchy	Returns to the Hierarchy window
Show Inherited	Toggles between showing all contents of object or class, and contents declared in current object or class

Object/class instance Inspector window

OOP

Object/class instance Inspector windows contain three panes, of which only the first two have local menus. (The third displays only the object type or class to which the instance belongs). Both local menus are the same, and contain the following commands:

Range	Selects array members to inspect
Change	Changes the value of highlighted item
Methods	Toggles whether methods or member functions are summarized in the middle pane

Show Inherited	Toggles between showing all contents of object or class and contents declared in current object or class
Inspect	Opens new Inspector window for highlighted item
Descend	Expands highlighted item into this Inspector window
New Expression	Inspects a new expression in this Inspector window
Type Cast	Type casts highlighted data item to new type
Hierarchy	Returns to the Object Hierarchy window

Text panes

This is the generic name for a pane that displays the contents of a text file. The blinking cursor shows your current position in the file. The following table lists all the commands:

Table 13.2
Text pane key commands

Key	Function
Ins	Marks text block
↑	Moves up one line
↓	Moves down one line
→	Moves right one column
←	Moves left one column
Ctrl →	Moves to next word
Ctrl ←	Moves to previous word
Home	Goes to start of line
End	Goes to last character on line
PgUp	Scrolls up one screen
PgDn	Scrolls down one screen
Ctrl-Home	Goes to top line of pane
Ctrl-End	Goes to bottom line of pane
Ctrl-PgUp	Goes to first line of file
Ctrl-PgDn	Goes to last line of file

If you are not using the control-key shortcuts, you can also use the WordStar-style control keys for moving around a text pane.

List panes

This is the generic name for a pane that lists information you can scroll through. A highlight bar shows your current position in the list. Here's a list of all the commands available to you.

Table 13.3
List pane key commands

Key	Function
↑	Moves up one item
↓	Moves down one item
→	Scroll right
←	Scroll left
Home	Goes to start of line
End	Goes to last character on line
PgUp	Scrolls up one screen
PgDn	Scrolls down one screen
Ctrl-Home	Goes to top line of list pane
Ctrl-End	Goes to bottom line of list pane
Ctrl-PgUp	Goes to first item in list
Ctrl-PgDn	Goes to last item in list
Backspace	Backs up one character in incremental match
Letter	Makes incremental search (select by typing)

You can also use the WordStar-style control keys for moving around a List pane.

Commands in input and history list boxes

The following table shows the commands available when you're inside an input or list box.

Table 13.4
Dialog box key commands

Key	Function
↑	Moves up one list item
↓	Moves down one list item
→	Moves right one character
←	Moves left one character
Ctrl →	Moves to next word
Ctrl ←	Moves to previous word
Home	Goes to start of line
End	Goes to last character on line
PgUp	Scrolls up one screen
PgDn	Scrolls down one screen
Ctrl-Home	Goes to top line of list pane
Ctrl-End	Goes to bottom line of list pane
Ctrl-PgUp	Goes to first item in list
Ctrl-PgDn	Goes to last item in list
Backspace	Deletes the character before the cursor
Enter	Accepts your input and proceed
Del	Deletes the character at the cursor
Esc	Cancels the dialog box and returns to menu
Ctrl-N	Completes partially typed name in input box

Window movement commands

Table 13.5
Window movement key commands

Key	Function
Ctrl-F5	Toggles window-positioning mode
↑	Moves window up one line
↓	Moves window down one line
→	Moves window right one column
←	Moves window left one column
Shift ↑	Resizes window; moves bottom up
Shift ↓	Resizes window; moves bottom down
Shift →	Resizes window; moves right side away from left
Shift ←	Resizes window; moves right side toward left
Home	Moves to left side of screen
End	Moves to right side of screen
PgUp	Moves to top line of screen
PgDn	Moves to bottom line of screen
Enter	Accepts current position
Esc	Cancels window-positioning command

Wildcard search templates

You can use wildcard search templates in two circumstances:

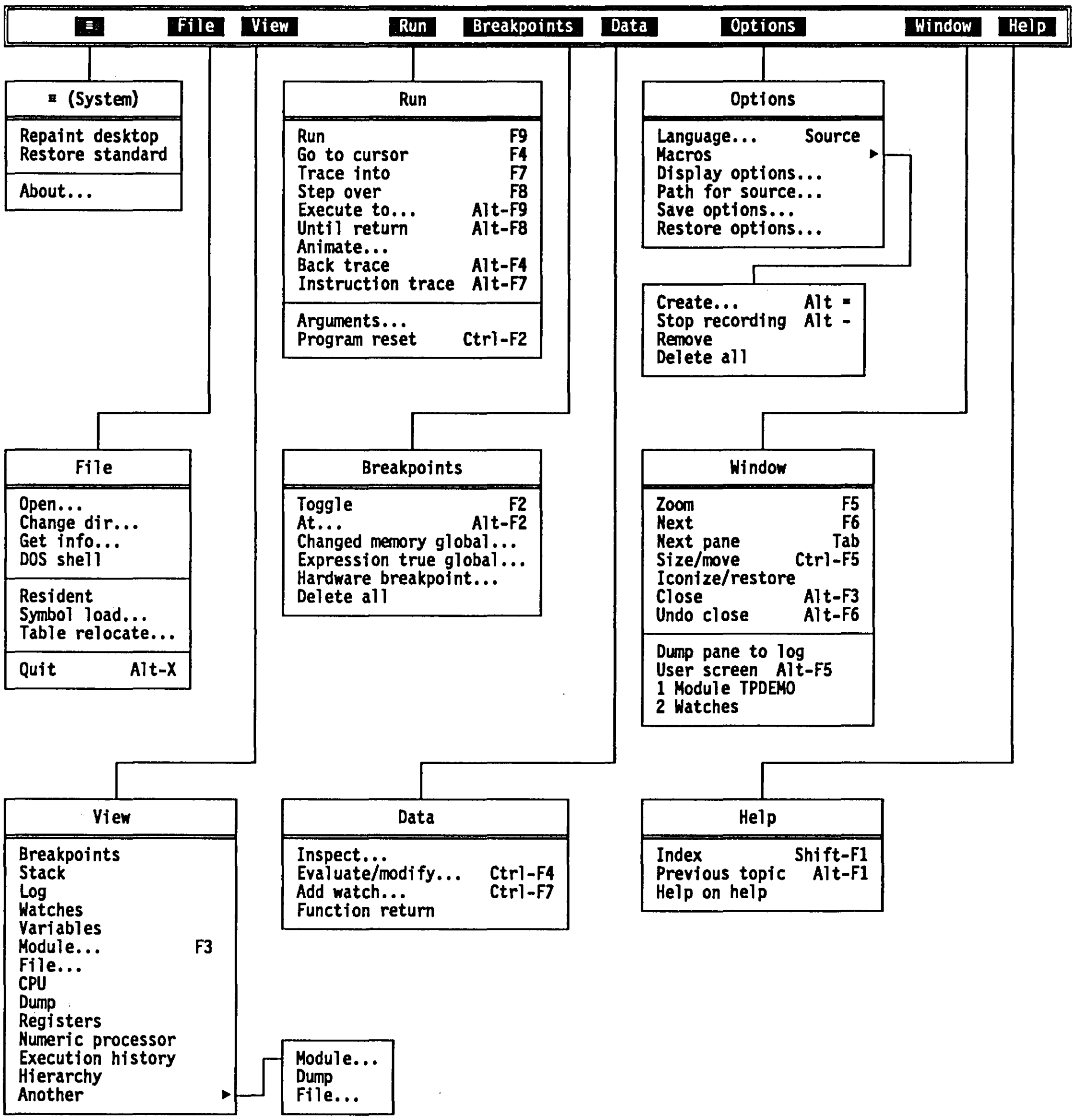
- when you enter a file name to load or examine
- when you enter a text search expression in a text pane

The ? (question mark) matches any single character in the search expression. The * (asterisk) matches 0 or more characters in the search expression.

Complete menu tree

Figure 13.1 shows the complete structure of Turbo Debugger's pull-down menus.

Figure 13.1: The Turbo Debugger menu tree



C H A P T E R

14

How to debug a program

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 objects. 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 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 Turbo Debugger 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 (C, Pascal, or assembler), 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 functions, a call to a function can leave unexpected contents in a variable or data structure:

```
char workbuf[20];
strcpy(workbuf,"all done\n");
convert("xyz");
printf(workbuf);
...
convert(char *p)
{
   strcpy(workbuf, p);
   while (*p)
      ...
}
```

Here, the correct thing to do would be to have the function use its own private work buffer.

Assuming initialized data

Don't assume that another routine has already set a variable for you:

```
char *workbuf;
addworkstring(char *s)
{
   strcpy(workbuf, s);      /* oops */
}
```

You should code a routine of this sort defensively by adding the statement

```
if (workbuf == 0) workbuf = (char *)malloc(20);
```

Not cleaning up

This sort of bug can crash your program by exhausting heap space:

```
crunch_string(char *p)
{
   char *work = (char *)malloc(strlen(p));
   strcpy(work, p);
   ...
   return(p);              /* whoops--work still allocated */
}
```

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 from the world of C programming:

```
for (n = 1; n < 10; n++)
{
   ...                     /* oops--only 9 times */
}
```

Here you can easily see the numbers 1 and 10, and you think that your loop goes from one to ten. (Better make that **<** into a **<=**.)

C-specific bugs

The *Turbo C User's Guide* has a section on pitfalls in C programming. However, this lesson on how to debug is a good place to reiterate those pitfalls and expand on them.

The Turbo C compiler is very good at finding C-specific bugs that other compilers don't warn you about. You can save yourself some debugging time by turning on all the warnings that the compiler is capable of generating. (See the *Turbo C User's Guide* for information on setting these warnings.)

What follows is by no means an exhaustive list of ways you can get in trouble with C. For some of these errors, the Turbo C compiler issues a warning message. Remember to examine the cause of any warning messages; they may be telling you about a bug in the making.

Using uninitialized autovariables

In C, an autovariable declared inside a function is undefined until you assign a value to it:

```
do_ten_times()
{
   int n;
   while (n < 10) {
        ...
        n++;
    }
}
```

This function executes the **while** loop an unpredictable number of times because *n* is not initialized to 0 before being used as a counter.

Confusing = and ==

C lets you both assign a value (=) and test for equality (==) within an expression; for example,

```
if (x = y) {
   ...
}
```

This inadvertently loads *y* into *x* and performs the statements in the **if** expression if the value of *y* is not 0. You almost certainly meant to say

```
if (x == y)
   ...
```

Confusing operator precedence

C has so many operators that it is hard to remember which ones are applied first when an expression is evaluated. One combination that often causes grief is the mixture of shift operators with addition or subtraction. For example,

```
x = 3 << 1 + 1
```

evaluates to 12, not 7, as you might expect if **<<** took effect before the **+**.

Bad pointer arithmetic

When you use a pointer to step through an array, be careful how you increment and decrement it. For example,

```
int *intp;
intp += sizeof(int);
```

does not increment *intp* to point to the next element of an integer array. Instead, *intp* is advanced by two array elements because in adding to or subtracting from a pointer, C takes into account the size of the item the pointer is pointing to. All you have to do to move the pointer to the next element is

```
intp++
```

Unexpected sign extension

Be careful about assigning between integers of different sizes:

```
int i = 0XFFFE;
long l;
l = i;
if (l & 0X80000000) {
   ...                  /* this DOES get executed */
}
```

One of C's strong points can cause you trouble if you are not aware of how it operates. C lets you assign freely between scalar values (**char, int**, and so on). When you copy an integer scalar into a larger scalar, the sign (positive or negative) is preserved in the larger scalar by propagating the sign (highest) bit throughout the high portion of the larger scalar. For example, an **int** value of –2 (0xfffe) becomes a **long** value of –2 (0xfffffffe).

Unexpected truncation

This problem is the opposite of the previous one:

```
int i;
long l = 0X10000;
i = l;
while (i > 0) {
   ...                  /* this does NOT get executed */
}
```

Here, the assignment of *l* to *i* resulted in the top 16 bits of *l* being truncated, leaving a value of zero in *i*.

Misplaced semicolons

The following code fragment may appear to be fine at first glance:

```
for (x = 0; x < 10; x++);
{
   ...                  /* only executed once */
}
```

Why does the code between the braces execute only once? Closer inspection reveals a semicolon (;) at the end of the **for** expression. This hard-to-find bug causes the loop to execute ten times, but does nothing. The subsequent block is then executed once. This is a nasty problem because you can't find it with the usual technique of examining the formatting and indenting of code blocks in your program.

Macros with side effects

The following problem is enough to make you swear off **#define** macros for life:

```
#define toupper(c) 'a'<= (c)&&(c)<='z' ? (c)-'a'-'A' : (c)
char c, *p;
c = toupper(*p++);
```

Here, *p* is incremented two or three times, depending on whether the character is uppercase. This type of problem is very hard to find, because the side effect is hidden within the macro definition.

Repeated autovariable names

Another hard one to find:

```
myfunc()
{
   int n;
   for (n = 5; n >= 0; n--)
   {
      int n = 10;
      ...
      if (n == 0)
      {
         ...                 /* never gets executed */
      }
   }
}
```

Here, the autovariable name *n* is reused in an inner block, hiding access to the one declared in the outer block. You must be careful about reusing variable names in this manner. You can get into trouble more easily than you might think, especially if you use a limited number of variable names for local loop counters (for example, *i*, *n*, and so forth).

Misuse of autovariables

This function means to return a pointer to the result:

```
int *divide_by_3(int n)
{
   int i;
   i = n / 3;
   return(&i);
}
```

The trouble is that by the time the function returns, the auto-variable is no longer valid and is likely to have been overwritten by other stack data.

Undefined function return value

If you don't end a function with the **return** keyword followed by an expression, it returns an indeterminate value; for example,

```
char *first_capital_letter(char *p)
{
   while (*p)
   {
      if ('A' <= *p && *p <= 'Z')
          return(p);
      p++;
   }

/* Oops--nothing returned here */

}
```

If there are no capital letters in the string, a garbage value is returned. You should put a `return(0)` as the last line of this function.

Misuse of break keyword

The **break** keyword exits from only a single level of **do, for, switch**, or **while** loops:

```
for (...)
{
   while (...) {
      if (...)
         break;      /* we want to exit for loop */
   }
}
```

Here, the **break** exits only from the while loop. This is one of the few cases where it is excusable to use the **goto** statement.

Code has no effect

Sometimes a typo results in perfectly compilable source code. However, it probably doesn't do what you want it to, and it may not do anything at all:

```
a + b;
```

Here, the intended line of code was *a* += *b*.

Pascal-specific bugs

Because of the strong type- and error-checking features of Pascal, there are few bugs specific to the language itself. However, because Turbo Pascal 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;
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.

Dangling 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)
```

and enable short-circuit Boolean evaluation, using the Turbo Pascal **{$B-}** option or the **O**ptions | **C**ompiler | **B**oolean command. That way, if *Temp* does equal **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 function very deep, and each of those procedures and functions can have its own declarations. Consider the following program:

```
program Confused;
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 214.

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

Like C, Pascal allows a "null" statement (one consisting only of a semicolon). Placed at the wrong spot, this can create all kinds of problems. Consider the following program:

```
program Test;
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, 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:

```
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 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 a flip 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.

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 }
```

This is all very nice, but it 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*.

There are two solutions to this type of problem. One is to turn off range-checking. However, that could end up introducing subtle bugs, especially if the code involved actually changes values. A better solution, shown earlier, is to select short-circuit Boolean evaluation, either by using the **O**ptions | **C**ompiler | **B**oolean command or by using the {**$B-**} directive. That way, if *I* > *Count*, the expression

```
List[I] <> Val
```

is never evaluated.

Assembler-specific bugs

Here are some of the common pitfalls of assembly language programming. You should refer to the *Turbo Assembler User's Guide* for a fuller explanation on these oft-encountered errors—and tips on how to avoid them.

Forgetting to return to DOS

In Pascal, C, and other languages, a program ends automatically and returns to DOS when there is no more code to execute, even if no explicit termination command was written into the program. Not so in assembly language, where only those actions that you explicitly request are performed. When you run a program that has no command to return to DOS, execution simply continues right past the end of the program's code and into whatever code happens to be in the adjacent memory.

Forgetting a RET instruction

The proper invocation of a subroutine consists of a call to the subroutine from another section of code, execution of the subroutine, and a return from the subroutine to the calling code. Remember to insert a **RET** instruction in each subroutine, so that the **RET**urn to the calling code occurs. When you're typing a program, it's easy to skip a **RET** and end up with an error.

Generating the wrong type of return

The **PROC** directive has two effects. First, it defines a name by which a procedure can be called. Second, it controls whether the procedure is a near or far procedure.

The **RET** instructions in a procedure should match the type of the procedure, shouldn't they?

Yes and no. The problem is that it's possible and often desirable to group several subroutines in the same procedure. Since these subroutines lack an associated **PROC** directive, their **RET** instructions take on the type of the overall procedure, which is not necessarily the correct type for the individual subroutines.

Reversing operands

To many people, the order of instruction operands in 8086 assembly language seems backward (and there is certainly some justification for this viewpoint). If the line

```
mov  ax,bx
```

meant "move AX to BX," the line would scan smoothly from left to right, and this is exactly the way in which many microprocessor manufacturers have designed their assembly languages. However, Intel took a different approach with 8086 assembly language; for us, the line means "move BX to AX," and that can sometimes cause confusion.

Forgetting the stack or reserving a too-small stack

In most cases, you are treading on thin ice if you don't explicitly allocate space for a stack. Programs without an allocated stack sometimes run, but there is no assurance that these programs will run under all circumstances. Most programs should have a **.STACK** directive to reserve space for the stack, and for each program that directive should reserve more than enough space for the deepest stack you can conceive of the program using.

Calling a subroutine that wipes out registers

When you're writing assembler code, it's easy to think of the registers as local variables, dedicated to the use of the procedure you're working on at the moment. In particular, there's a tendency to assume that registers are unchanged by calls to other procedures. It just isn't so—the registers are global variables, and each procedure can preserve or destroy any or all registers.

Using the wrong sense for a conditional jump

The profusion of conditional jumps in assembly language (**JE, JNE, JC, JNC, JA, JB, JG,** and so on) allows tremendous flexibility in writing code—and also makes it easy to select the wrong jump for a given purpose. Moreover, since condition-handling in assembly language requires at least two separate lines, one for the comparison and one for the conditional jump (it requires many more lines for complex conditions), assembly language condition-handling is less intuitive and more prone to errors than condition-handling in C and Pascal.

Forgetting about REP string overrun

String instructions have a curious property: After they're executed, the pointers they use wind up pointing to an address 1 byte away (or 2 bytes for a word instruction) from the last address processed. This can cause some confusion with repeated string instructions, especially **REP SCAS** and **REP CMPS.**

Relying on a zero CX to cover a whole segment

Any repeated string instruction executed with CX equal to zero does nothing. Period. This can be convenient in that there's no need to check for the zero case before executing a repeated string instruction; on the other hand, there's no way to access every byte in a segment with a byte-sized string instruction.

Using incorrect direction flag settings

When a string instruction is executed, its associated pointer or pointers—SI or DI or both—increment or decrement. It all depends on the state of the direction flag.

The direction flag can be cleared with **CLD** to cause string instructions to *increment* (count up) and can be set with **STD** to cause string instructions to *decrement* (count down). Once cleared or set, the direction flag stays in the same state until either another **CLD** or **STD** is executed, or until the flags are popped from the stack with **POPF** or **IRET**. While it's handy to be able to program the direction flag once and then execute a series of string instructions that all operate in the same direction, the direction flag can also be responsible for intermittent and hard-to-find bugs by causing the behavior of string instructions to depend on code that executed much earlier.

Using the wrong sense for a repeated string comparison

The **CMPS** instruction compares two areas of memory; the **SCAS** instruction compares the accumulator to an area of memory. Prefixed by **REPE**, either of these instructions can perform a comparison until either CX becomes zero or a not-equal comparison occurs. Unfortunately, it's easy to become confused about which of the **REP** prefixes does what.

Forgetting about string segment defaults

Each of the string instructions defaults to using a source segment (if any) of DS, and a destination segment (if any) of ES. It's easy to forget this and try to perform, say, a **STOSB** to the data segment, since that's where all the data you're processing with nonstring instructions normally resides.

Converting incorrectly from byte to word operations

In general, it's desirable to use the largest possible data size (usually word, but dword on an 80386) for a string instruction, since string instructions with larger data sizes often run faster.

There are a couple of potential pitfalls here. First, the conversion from a byte count to a word count by a simple

```
shr cx,1
```

loses a byte if CX is odd, since the least-significant bit is shifted out.

Second, make sure you remember **SHR** divides the byte count by two. Using, say, **STOSW** with a byte rather than a word count can wipe out other data and cause problems of all sorts.

Using multiple prefixes

String instructions with multiple prefixes are error-prone and should generally be avoided.

Relying on the operand(s) to a string instruction

The optional operand or operands to a string instruction are used for data sizing and segment overrides only, and do not guarantee that the memory location referenced is accessed.

Wiping out a register with multiplication

Multiplication—whether 8 bit by 8 bit, 16 bit by 16 bit, or 32 bit by 32 bit—always destroys the contents of at least one register other than the portion of the accumulator used as a source operand.

Forgetting that string instructions alter several registers

The string instructions, **MOVS, STOS, LODS, CMPS,** and **SCAS,** can affect several of the flags and as many as three registers during execution of a single instruction. When you use string instructions, remember that SI, DI, or both either increment or decrement (depending on the state of the direction flag) on each execution of a string instruction. CX is also decremented at least once, and possibly as far as zero, each time a string instruction with a **REP** prefix is used.

Expecting certain instructions to alter the carry flag

While some instructions affect registers or flags unexpectedly, other instructions don't even affect all the flags you might expect them to.

Waiting too long to use flags

Flags last only until the next instruction that alters them, which is usually not very long. It's a good practice to act on flags as soon as possible after they're set, thereby avoiding all sorts of potential bugs.

Confusing memory and immediate operands

An assembler program may refer either to the offset of a memory variable or to the value stored in that memory variable. Unfortunately, assembly language is neither strict nor intuitive about the ways in which these two types of references can be made, and as a result, offset and value references to a memory variable are often confused.

Causing segment wraparound

One of the most difficult aspects of programming the 8086 is that memory isn't accessible as one long array of bytes, but is rather made available in chunks of 64K relative to segment registers. Segments can introduce subtle bugs; if a program attempts to access an address past the end of a segment, it actually ends up wrapping back to access the start of that segment instead.

Failing to preserve everything in an interrupt handler

Every interrupt handler should explicitly preserve the contents of all registers. While it is valid to preserve explicitly only those registers that the handler modifies, it's good insurance just to push all registers on entry to an interrupt handler and pop all registers on exit.

Forgetting group overrides in operands and data tables

Segment groups let you partition data logically into a number of areas without having to load a segment register every time you want to switch from one of those logical data areas to another.

Unfortunately, there are a few problems with the way the Microsoft Macro Assembler (MASM) handles segment groups, so until Turbo Assembler came along, segment groups were quite a nuisance in assembler. They were, however, an unavoidable nuisance, for they are required in order to link assembler code to high-level languages such as C.

In MASM Quirks mode, Turbo Assembler emulates MASM, warts and all. This means that in MASM Quirks mode, Turbo Assembler has the same problems with segment groups that MASM has. If you're not planning to use MASM Quirks mode, read no more, but if you are going to use MASM Quirks mode, refer to the *Turbo Assembler User's Guide* for more information.

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 being given a clean bill of health.

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 215).

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

This 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 is a version of the demonstration program used in Chapter 3 (TCDEMO.C or TPDEMO.PAS), except this one has some deliberate bugs in it.

Make sure that your current directory contains the two files needed for the debugging demonstration. If you're a C programmer, you'll need TCDEMOB.C and TCDEMOB.EXE. If you're debugging a Pascal program, you'll need TPDEMOB.PAS and TPDEMOB.EXE. (The *B* in these file names stands for "buggy.")

Go ahead and compile the source code program to generate your .EXE file. (If you are compiling TCDEMOB.C, open it in the integrated development environment and set the **O**ptions | **C**ompiler | **O**ptimization | **U**se Register Variables switch to *Off* before you compile.)

C debugging session

This section uses a Turbo C program as its example. If you're a Pascal programmer, refer to page 238 for the sample debugging session using a Turbo Pascal program.

Looking for errors

Before we start the debugging session, let's run the buggy demo program to see what's wrong with it. To start the program, type

```
TCDEMOB
```

You are prompted for lines of text. Enter two lines of text

```
one two three
four five six
```

A final empty line ends your input. TCDEMOB then prints out its analysis of your input:

```
Arguments:
Enter a line (empty line to end): one two three
Enter a line (empty line to end): four five six
Enter a line (empty line to end):
Total number of letters = 7
Total number of lines = 6
Total word count = 2
Average number of words per line = 0.3333333
'E' occurs 1 times, 0 times at start of a word
'F' occurs 1 times, 1 times at start of a word
```

```
'N' occurs 1 times, 0 times at start of a word
'O' occurs 2 times, 1 times at start of a word
'R' occurs 1 times, 0 times at start of a word
'U' occurs 1 times, 0 times at start of a word
There is 1 word 3 characters long
There is 1 word 4 characters long
```

Notice there are erroneous numbers for the total number of words, letters, and word count. Later on, the letter and word frequency tables seem to be based on an erroneous letter and word count. This is an all-too-typical situation—the program must have more than one thing wrong. This happens frequently in the early stages of debugging a program.

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, each input line is broken down into words, then analyzed, and finally, after all the lines have been entered, the tables are displayed. Since the word and letter counts are off as well as the tables, it's a good bet that something is wrong during the initial breaking down and counting phase.

Now is the time to start debugging, *after* you've thought about the problem for a moment and decided on a rough plan of attack. Here, the strategy is to examine the routine *makeintowords*, to see if it is correctly chopping the line into null-terminated words, and then see if *analyzewords* is correctly counting the analyzed line.

Starting Turbo Debugger

To start the debugging session, type

```
TD TCDEMOB
```

Turbo Debugger loads the buggy demo program and then displays its startup screen. If you wish to exit from the tutorial session and return to DOS, press *Alt-X* at any time. If you get hopelessly lost, you can reload the demonstration program at any time and start at the beginning by pressing *Ctrl-F2*. (Note that this doesn't clear breakpoints or watches.)

Since the first thing you want to do is to check that *makeintowords* is working correctly, run the program up to that routine and then check it. There are two approaches you can use: Either *step* through *makeintowords* as it executes, making sure that it does the right thing, or stop the program *after makeintowords* has done its stuff and see if it did the right thing.

Since *makeintowords* has a clearly defined task and it's easy to determine whether it's working correctly by inspecting the output buffer it produces, let's opt for the second approach. To do this, move down to line 42 and press *F4* to run to this line. When the program screen appears, type

```
one two three
```

and press the *Enter* key.

Inspecting

You are now stopped at the source line after the call to *makeintowords*. Look at the contents of *buffer* to see if the right thing happened. Move the cursor up a line, place it under the word *buffer*, and press *Alt-F10 I* (for Inspector) to open an Inspector window to show the contents of *buffer*. Use the arrow keys to scroll through the elements in the array. Notice that *makeintowords* has indeed put a single null character (0) at the end of each word as it is meant to. This means that you should execute more of the program and see if *analyzewords* is doing the right thing. First, remove the Inspector window by pressing *Esc*. Then, press *F7* twice to execute to the start of *analyzewords*.

Breakpoints

Check that *analyzewords* has been called with the correct pointer to the buffer by moving the cursor under *bufp* and pressing *Alt-F10 I*. You can see that *bufp* indeed points to the null-terminated string `'one'`. Press *Esc* to remove the Inspector window. Since there seems to be a problem with counting characters and words, let's put a breakpoint at the places where a character and a word are counted:

1. Move to line 93 and press *F2* to set a breakpoint.
2. Move to line 97 and set another breakpoint.
3. Finally, set a breakpoint on line 99 so you can look at the character count this function returns.

Setting multiple breakpoints like this is a typical way to learn about whether things are happening in the right order in a program, and lets you check on important data values each time the program stops at a breakpoint.

The Watches window

Run the program by pressing *F9*. The program stops when it reaches the breakpoint on line 93. Now you want to look at the value of *charcount*. Since you'll want to check it each time you hit a breakpoint, this is an ideal time to use the **Watch** command to place it in the Watches window. Move the cursor under *charcount* and press *Alt-F10 W*. The Watches window at the bottom of the screen now displays the current value of 0. To make sure that the character is being counted properly, execute a single line by pressing *F7*. The Watches window now shows that *charcount* is 1.

The Evaluate/ Modify dialog box

Run the program again by pressing *F9*. You are now back at line 93 for another character. Press *F9* again twice to read the last letter on the word and the terminating null. *charcount* now correctly shows 3, and the *wordcounts* array is about to be updated to count a word. Everything is fine so far. Press *F9* again to start processing the next word in the buffer. AHA! Something is wrong.

You expected the program to stop again on line 93 as it processed the next word, but it didn't. It went straight to the statement that returns from the function. The only way to end up on line 99 is if the **while** loop that started on line 83 no longer has a true test value. This means that **bufp* != 0 must evaluate to false (that is, 0).

To check this, move back to line 83 and mark the entire expression **bufp* != 0 by putting the cursor under the *, pressing *Ins*, and moving the cursor to the final `'0'` before the `')'`. Now evaluate this expression by opening the **D**ata | **E**valuate Modify dialog box and pressing *Enter*, and choosing the Eval button to accept the marked expression. The value is indeed 0. Press *Esc* to return to the Module window.

Eureka!

Now here comes the analytical leap that causes you to "solve" the bug. The reason *bufp* points to a 0 is because that is where the inner **while** loop starting on line 86 left it at the end of a word. To continue to the next word, you must increment *bufp* past the 0 that ended the previous word. To do this, you need to add a *"bufp++"* statement before line 97. You could recompile your program with this statement added, but Turbo Debugger lets you "splice" in expressions by using a fancy sort of breakpoint.

To do this, first reload the program by pressing *Ctrl-F2* so you can test with a clean slate. Now remove all the breakpoints you set in the previous session by typing *Alt-B D*. Go back to line 97 and set a breakpoint again by pressing *F2*. Now, open a Breakpoints window by pressing *Alt-V B*. Set this breakpoint to execute the expression *bufp++* each time it is encountered:

1. Choose **V**iew | **B**reakpoints.
2. Open the Breakpoints window local menu by pressing *Alt-F10*.
3. Choose **S**et Options to open the Breakpoint Options dialog box.
4. Set the Action radio buttons to Execute.
5. Press *Tab* to get to the Action Expression prompt.
6. Enter `bufp++`.
7. Press *Esc* to close the dialog box and *Alt-F3* to return to the Module window.

Now run the program. Enter the usual two input lines

```
one two three
four five six
```

Press *Enter* at the third prompt, and when the program has terminated, press *Alt-F5* to look at your output on the User screen.

You'll notice that things have improved considerably. The total number of words and lines seem to be wrong, but the tables are correct. Stop at the beginning of the *printstatistics* routine and see if it is given the correct values to print. First reload the program by pressing *Ctrl-F2* to retest. Then go to line 104 and press *F4* to execute to there. Move the cursor to the *nlines* argument and press *Alt-F10 I* to look at its value. Note that the value is 6 where it should be 2.

Now go back to where *nlines* is called from in **main** and look at the its value there. Move the cursor to line 36, place it under *nlines*, and press *Alt-F10 I* to look at the value. The value of *nlines* in **main** is 2, which is correct! If you go down to line 46, you will notice that the two arguments *nwords* and *nlines* have been reversed. There is no way that the compiler could have known that you meant to have them the other way around.

If you correct these two bugs, the program will run correctly. The files TCDEMO.EXE is a corrected version that you may run if you are curious.

Pascal debugging session

This section uses a Turbo Pascal program as its example. If you're a C programmer, you should look at the preceding section, starting on page 233, which takes you through a session using a Turbo C program.

Looking for errors

Before we start the Pascal debugging session, let's run the buggy Pascal demo program to see what's wrong with it. The program is already compiled and on your distribution disk.

To start the program, enter the program name and pass it three command-line arguments:

```
TPDEMOB first second third
```

You'll be prompted for lines of text. Enter two lines of text exactly as follows:

```
ABC DEF GHI
abc def ghi
```

A final empty line ends your input. TPDEMOB then prints out its analysis of your input:

```
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

Program name: C:\td\tpdemob.ex♣
Command line parameters: firs☻ secon♪ third
```

There are five 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.
5. The last character of each command-line parameter entered was lost and random characters are being displayed (although the last parameter is okay).

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*. Finally, in a completely independent subprogram, procedure *ParmsOnHeap* builds a linked list of command-line parameters on the heap and then traverses and displays that list at the end of the program.

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.

Finally, once you've scrutinized the word and letter counting parts of the program, take a look at *ParmsOnHeap* to find and fix the last (fifth) bug.

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 Turbo Debugger

To start the debugging sample session, load the debugger and give it the same command-line parameters you gave it earlier:

```
TD TPDEMOB first second third
```

Turbo Debugger loads the buggy demo program and displays the startup screen. If you wish to exit from the tutorial session and return to DOS, 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 231, press *F4* (**G**o to Cursor command), and 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`).

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 133 and press *F4* (**G**o 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.

Get back to the line that called *IsLetter* by another navigation technique: Press *Alt-F8*, which runs past the end statement of the current procedure or function. Since the second line of input you originally entered, `abc def ghi`, contained only lowercase letters, each character was treated as whitespace and skipped. This throws off both the letter counts and the word count, and solves the mysteries of bugs #1 and #4.

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* and enter the following expression:

```
IsLetter('a') = IsLetter('A')
```

A and *a* are both letters, but the evaluation False 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.)

Inspecting

Two bugs down, three 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 234 and press *F4* to run to the cursor position.

TPDEMOB 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 (you can also use the hot key, *Ctrl-I*). *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, type *Alt-R P* to end the current debug session, press *F9* to run the program from the beginning, and press *Enter* at TPDEMOB's string prompt. The program terminates and an error box displays a run-time error. You should modify this statement to read

```
if NumLines <> 0 then
  AvgWords := NumWords / NumLines
else
  AvgWords := 0;
```

So much for bugs #2 and #2b. 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 *Alt-F10 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. This is a very powerful way of examining your data structures, and will be especially handy when you traverse *ParmsOnHeap*'s linked list later on.

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 pressing *Alt-R P* (to reset the program). Then 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. Turbo Debugger should be stopped at *ShowResults*.

ShowResults uses a nested procedure, *ShowLetterInfo*, to display the letter tables. Move the cursor down to line 103, 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 63. Position the cursor over *FromLet* and *ToLet* and use *Alt-F10 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 64. 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 64, the current value of *ch*, *M*, is output and the program moves on. Removing that extra semicolon eliminates bug #3.

Just one more bug...

It's time to track down that strange bug with the command-line parameters. To refresh your memory, the last character of all but the last command-line parameter was garbage. Perhaps the string length byte was wrong, or perhaps the string data was overwritten by some later assignment.

Use the Watches window to find out. Press *Alt-F9*, type `parmsonheap` and press *Enter*. The **for** statement loops through all the command-line parameters, constructing a linked list and copying each string onto the heap as it goes. One pointer, *Head*, points to the beginning of the list; *Tail* points to the last node in the list; and *Temp* is used as temporary storage to allocate and initialize a new node. Since the string data is corrupted, press *Ctrl-F7* and add the following expression to the Watches window:

```
Tail^.Parm^
```

This keeps track of the string data stored in the last node in the list. Of course, this value will be garbage until *Tail* is initialized on line 207.

Rather than step through line-by-line, just keep an eye on the Watches window at the end of each iteration. Move the cursor to line 208 and press *F2* to set a breakpoint there. Now press *F9* to run to that breakpoint. If you're using DOS 3.x, you'll see the full path to TPDEMOB.EXE in the Watches window. (If you're using DOS 2.x, you'll see an empty string; in that case, just press *F9* again and then go on.) The string data looks just fine.

Press *F9* to execute the loop another time. Again, the data looks okay. Now you know that the string is being copied onto the heap correctly. You can use the Inspector window to find out whether it's been corrupted yet. Move the cursor over *Head* on line 203 and press *Alt-F10 I*.

Look at the value referenced by *Parm* by pressing ↓, followed by *Enter*. You're looking at the first node in the list, and its string data is already corrupted. If you press *Esc*, ↓, and then press *Enter* again, you'll open an Inspector window onto the second node in the list. Press ↓, followed by *Enter*, to inspect its string data. It's intact, and, in fact, is the same node referenced by the *Tail* pointer. Something is definitely clobbering the tail end of the string data.

Keep your eye on the Watches window while you use *F7* to step through the loop. The call to *GetMem* on line 199 is the culprit; before that call, *Tail^.Parm^* is equal to *first*. Immediately after the call to *GetMem*, the last character in *Tail^Parm^* is trashed.

What's happening? For each command-line parameter, the **for** loop allocates first a record, then the string data, then the next record, and so on. The *GetMem* call on line 199 should allocate enough for the length of the string plus the length byte, but you can see it does not add 1 to *Length*(s). Though the string assignment on line 200 succeeds in doing the copy, it actually uses 1 more byte than was allocated to it. Thus, the last character of the string is overlapped by the first byte of the next record allocated when a call is made to *New(Temp)*. The last parameter escapes unscathed because it's not followed by another *ParmRec*.

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 TPDEMO.PAS, the bug-free version of this program, discussed in Chapter 3.

C H A P T E R

15

Virtual debugging on the 80386 processor

Turbo Debugger lets you use the full power of systems that have the 80386 processor. Virtual debugging lets the program you're debugging use the full address space below 640K, just as if no debugger were loaded. (Turbo Debugger is loaded into extended memory, above the 1MB address point.)

You debug exactly as you would normally use Turbo Debugger, except that once the TDH386 device driver is loaded, your program loads and runs at exactly the same address whether or not it's being debugged. Virtual debugging is extremely useful both for debugging programs that are large, and for finding bugs that go away if the program is loaded higher in memory, as it is when it is being debugged normally.

Virtual debugging also lets you watch for reads or writes to arbitrary memory or I/O locations, all at full or nearly full processor speed. This gives you some of the power of a hardware debugger at no additional cost.

80286 users!

If you have an 80286 processor, you can make more memory available than you would normally have with Turbo Debugger by using the protected-mode debugger, TD286. See Chapter 16 for more information.

Equipment required for virtual debugging

You must have a computer based on the 80386 processor in order to use the virtual debugger. You must also have 640K of available extended memory. If you have used up your extended memory for RAM disks, caches, and so forth, you may want to make a special CONFIG.SYS or AUTOEXEC.BAT file that removes some of these programs when you want to use virtual debugging.

Installing the virtual debugger device driver

Before starting the virtual debugger, you must make sure that you have installed its device driver in your CONFIG.SYS file. Do this by including a line similar to the following in CONFIG.SYS:

```
DEVICE = TDH386.SYS
```

If you have placed the TDH386.SYS device driver somewhere other than in the root directory, make sure that you include that directory path as part of the device driver file name.

Normally, the virtual debugger lets you have up to 256 bytes of DOS environment strings. If this is not enough, or if you don't need that much and would like to conserve as much memory as possible, use the **–e** option in CONFIG.SYS to set the number of bytes of environment. For example,

```
DEVICE = TDH386.SYS -e2000
```

reserves 2000 bytes for your DOS environment variables.

Starting the virtual debugger

You start the virtual debugger much as you would normally start Turbo Debugger, with a command line like this:

```
TD386 [options] program [program options]
```

In other words, you simply enter `TD386` instead of `TD`. TD386 then takes care of finding the Turbo Debugger executable program and loading it into extended memory.

If you have other programs or device drivers that use extended memory, such as RAM disks, caches, or whatever, you must tell

TD386 how much extended memory to set aside for these other programs. Do this by using the **–e** command-line option. Follow the **–e** with the number of kilobytes (K) of extended memory used by the other programs. For example,

```
TD386 -e512 myprog
```

This command line informs TD386 that you want to reserve the first 512K of extended memory for other programs.

Normally, if your system supports the XMS standard, it is not necessary to inform TD386 how much memory to set aside for programs in extended memory; the programs have already passed that information to TD386. You need to use **–e** only with programs (such as VDISK) that don't communicate with the XMS standard.

Since you probably always reserve the same amount of extended memory for other programs, TD386 gives you a way to permanently set the amount of extended memory to reserve. Use the **–w** option with the **–e** option to specify that you want the **–e** value to be permanently set in the TD386 executable program file.

You'll then be prompted for the name of the executable program. If you are running on DOS 3.0 or later, the prompt indicates the path and file name that you executed TD386 from. You can accept this name by pressing *Enter*, or you can enter a new executable file name. The new name must already exist and be a copy of the TD386 program that you have already made.

If you are running on version 2.x of DOS, you will have to supply the full path and file name of the TD386 executable program.

Here is a complete list of command-line options for TD386.EXE:

–?, –h	Accesses help on TD386.
–b	Lets you break out of programs with *Ctrl-Break*, even when interrupts are disabled.
–e####	Specifies the number of kilobytes of extended memory being used by other programs or by the program you're debugging. (You don't need this option if your system supports the XMS standard.)
–f####	Enables EMS emulation through paging (in extended memory) and sets the page frame segment to #### (in hex). The last three digits

must be 000 (like C000 or E000). Note that this option only applies to Turbo Debugger's EMS calls. If you don't use this option when you load TD386, TD386 will not be able to use EMS. If you cannot load your symbol table, try using the **–f** option to force TD386 to borrow from extended memory.

No real EMS:	–fD000
Real EMS at D000:	–fE000
Real EMS at E000:	–fD000

–f- Disables EMS emulation (presumably to override a previous command-line option).

–w Modifies TD386.EXE with the new default value of **–e** or **–f**. You can enter a new executable file name that does not already exist, and TD386 will create the new executable file.

Note that TD386.EXE options must appear first in the command line before any Turbo Debugger options or the program name. For example,

```
TD386 -e1024 -fD000 -w
```

reserves 1024K of extended memory, enables EMS emulation with a page frame of D000, and modifies TD386.EXE with these values.

For a list of all the command-line options available for TD386.EXE, just type `TD386 -?` or `TD386 -h` and press *Enter.*

Note: If you have an 80386-based machine and want to read the command-line options for TD386.EXE, TDH386.SYS must be loaded.

Differences between normal and virtual debugging

Most things work exactly the same whether you are debugging normally or using the 80386 virtual debugging capability. The following items behave differently:

- When you use the **File | DOS Shell** command to run a DOS command, the program you're debugging is never swapped to disk. This means you may not always have enough memory to run other programs from the DOS prompt.

- Your program can use nearly all of the 80386 instructions, with the exception of the privileged protected-mode instructions: **CLTS, LMSW, LTR, LGDT, LIDT, LLDT.**
- Even though you can use all the 80386 extended addressing modes and 32-bit registers during virtual debugging, you can't access memory above the 1MB point. If you try to do so, an exception interrupt will be generated, and Turbo Debugger will regain control.
- You can't use virtual debugging if you're already running a program or device driver that uses the virtual and protected modes of the 80386 processor. This includes programs such as:
  - DesqView operating environment
  - Microsoft Windows-386 operating environment
  - QEMM.SYS, the QuarterDeck EMS simulator
  - CEMM.SYS Compaq EMS simulator
  - 386^MAX

  If you normally use one of these or similar programs, you will have to stop them or unload them before using TD386.
- If you are using virtual debugging, TD386 can catch exceptions generated by your program. If an exception occurs, your program stops, and TD386 reports the exception that occured. The error message that appears indicates the nature of the exceptions, and the arrow in the CPU window Code pane—or the cursor in the Module window—marks the instruction that caused the exception.
- You should not get an unexpected interrupt. If you do, contact Borland technical support.

TD386 error messages

TD386 generates one of the following messages when it can't start, and then returns to the DOS prompt. You must correct the condition before you can start TD386 successfully.

TD386 error: 80386 device driver missing or wrong version
You must install the TDH386.SYS device driver in your CONFIG.SYS file before you invoke TD386 from the DOS command line.

TD386 error: Can't enable the A20 address line
TD386 can't access the memory above 1MB. This may happen if you're running on a system that is not exactly IBM compatible.

TD386 error: Can't find TD.EXE
TD386 could not find TD.EXE.

TD386 error: Couldn't execute TD.EXE
TD386 could not run TD.EXE.

TD386 error: Environment too long; use –e#### switch with TDH386.SYS
You need to change the **–e** option as described on page 248.

TD386 error: Not enough Extended Memory available
TD386 ran out of memory. You need to get more memory for your machine or free up memory (by reducing a RAM disk, for example).

TD386 error: Wrong CPU type (not an 80386)
You are not running on a system with an 80386 processor.

The following errors might occur if you're trying to modify TD386 with the **–w** option:

TD386 error: Cannot open program file

TD386 error: Cannot read program file

TD386 error: Cannot write program file

TD386 error: Program file corrupted or wrong version

TDH386.SYS error messages

There are only two possible error messages associated with the TDH386.SYS driver:

Wrong CPU type: TDH386 driver not installed

Invalid command line: TDH386 driver not installed

C H A P T E R

16

Protected-mode debugging with TD286

The TD286 protected-mode debugger takes advantage of the capabilities of the 80286 processor to free more memory for the program you are debugging. TD286 puts the Turbo Debugger program into extended memory above the 1MB address point, and leaves a relatively small loader in the lower 640K. This gives you more room for the program you are debugging and its symbol table.

Use Turbo Debugger exactly as you normally would. The only difference is that your program has more memory to run in.

80386 users!

If you have an 80386 processor, you can get even more capabilities and memory savings by using the TD368 virtual debugger. See Chapter 15 for more information.

Equipment required for the protected-mode debugger

To use the TD286 protected-mode debugger, you must have a computer based on the 80286 or 80386 processor. You must also have at least 640K of available extended memory.

Installing the protected-mode debugger

Before you use TD286 for the first time, you must run the TD286INS configuration program to let TD286 determine some hardware characteristics of the system you are running on. To configure TD286, run the configuration program by entering `TD286INS` at the DOS prompt.

TD286INS asks you to press *Spacebar* a number of times as it determines the characteristics of your hardware. If at any point your system hangs and the program does not proceed, just reboot and restart the configuration program. The configuration program knows where it had a problem and continues with the next phase of its testing.

Once TD286INS runs to completion, TD286 is ready to use.

Starting the protected-mode debugger

You start the protected-mode debugger with this command-line syntax:

```
TD286 [options] program [program options]
```

TD286 has the same command-line options as regular Turbo Debugger, with the exception that it does not allow the **–y** option that sets the overlay code pool size. This option is not necessary because TD286 does not use overlays.

Differences between Turbo Debugger and protected-mode

There are a few things you can do in regular Turbo Debugger that you can't do with TD286:

- When you use the **File | DOS** Shell command to run a DOS command, the program you are debugging is not swapped to disk. This means that you may not always have enough memory to run other programs from the DOS prompt.

- You can't use TD286 to debug programs that run in protected mode, or use a DOS extender that conflicts with that used by TD286.

Running TD286 on different machines

TD286 knows the hardware characteristics of dozens of different machines. When you run TD286INS and it reports "**Machine already in file's database**" your machine is already known to TD286 and no modification is necessary.

If TD286INS does execute its tests, it will store your machine's hardware characteristics in TD286 and create a file with the .DB extension. This file should be sent back to Borland or uploaded onto one of our forums on Compuserve so that future versions of TD286 will automatically know your computer's hardware characteristics. TD286 can store the characteristics of 10 machines other than the ones it starts with.

C H A P T E R

17

Debugging TSRs and device drivers

With Turbo Debugger 2.0 you can debug terminate and stay resident (TSR) programs and device drivers, as well as conventional executable files. You can also run Turbo Debugger itself as a TSR, while you perform other operations at DOS level or run other programs.

Turbo Debugger 2.0 has three new commands on the file menu that are specifically designed to be used for debugging TSRs and device drivers. These are the **F**ile | **R**esident, File | **S**ymbol Load, and **F**ile | **T**able Relocate commands.

This chapter gives a brief explanation of what TSRs and device drivers are, and provides information on how to debug them with Turbo Debugger 2.0.

What's a TSR?

TSR stands for "terminate and stay resident." TSRs are programs that stay in RAM after they are finished running. SideKick and SuperKey are TSRs; they stay in RAM all the time and are invoked using special hot keys. Other TSRs are invoked from programs that issue an appropriate software interrupt. Turbo C provides a function, **geninterrupt**, that issues such software interrupts.

TSRs consist of two parts: a *transient portion* and a *resident portion.* The transient portion is responsible for loading the resident portion into RAM, and for installing an interrupt handler that determines how the TSR is invoked. If the TSR is to be invoked through a software interrupt, the transient portion places the address of the resident portion of the code in the appropriate interrupt vector. If the TSR is to be invoked through a hot key, the resident portion must modify the DOS interrupt handler for keyboard presses.

When the transient portion is finished executing, it invokes a DOS function that allows a portion of the .EXE file to stay resident in RAM after execution is terminated—hence the phrase "terminate and stay resident." The transient portion of the TSR knows the size of the resident portion as well as the resident portion's location in memory, and passes this information along to DOS. DOS then leaves the specified block of memory alone, but is free to overwrite the unprotected portion of memory. Thus the resident portion stays in memory, while the transient portion can be overwritten.

The trick to debugging TSRs is that you want to be able to debug the resident portion as well as the transient portion. When the .EXE file executes, the only code that is executed is the transient portion of the TSR. So when you run Turbo Debugger as usual, by specifying a file name, the only code you see executed is the transient portion, as it installs the resident portion and its interrupt handlers. In order to debug the resident portion, you must set a debugger breakpoint and make Turbo Debugger itself go resident. More about this later.

Debugging a TSR

Debugging the transient portion of a TSR is the same as debugging any other file. It is only when you start to debug the resident portion that anything novel happens.

Here is how you debug a TSR program:

1. Compile or assemble the TSR, being sure to incorporate symbolic (debugging) information. Use the TASM **/ZI** or TCC **–v** command-line option, for example, or TPC **/V**.
2. If you have to link the TSR, use the **/v** option to incorporate symbolic information. You can use the TDSTRIP **–s** option to move the symbolic information into a separate file, though you don't have to if the program is an .EXE file.

3. Now load the TSR program into Turbo Debugger and run the transient portion, using the **Run | Run** command as usual. Go ahead and debug the transient portion in the usual way. When you finish running the transient portion, the resident portion is installed in RAM. The trick now is to debug the resident portion.
4. Set a breakpoint at the beginning of the resident portion of your code, using *F2*. You can instead set breakpoints at some other positions in the resident portion, if you want.
5. Choose the **File | Resident** command to make Turbo Debugger itself go resident. This has nothing to do with making your TSR memory-resident; it makes itself go resident when you run it in Turbo Debugger, just as it would if you had run it from the command line. The only reason you are making Turbo Debugger go resident is so you can go back to DOS and invoke your TSR, making its resident portion start executing.
6. When you are back at the DOS command line, execute the resident portion of your TSR by pressing its hot key or doing whatever else you do to invoke it. Execute your program as usual.
7. When your program hits the breakpoint, Turbo Debugger comes back up, with your TSR displayed at the appropriate point. Now you can start debugging the resident part of your code. (You can also re-enter Turbo Debugger from DOS by pressing *Ctrl-Break* twice.)

A second method of debugging a TSR's resident portion is to execute the TSR from the DOS command line, then use Turbo Debugger to debug the area of RAM containing the TSR.

To use this method, you need the utilities TDMEM, which displays a map of how your system's RAM memory is used, and TDDEV, which gives the segment address where your TSR's resident portion is loaded.

To use this method:

1. Follow Steps 1 through 2 of the first method to compile or assemble your code, and to strip off its symbol table if necessary and place it in a .TDS file. If necessary for your application, run TDSTRIP with the **–c** option as well, to convert your TSR from an .EXE to a .COM file.

2. Execute your TSR from the DOS command line by typing its name. For example, if your TSR is called TSR.EXE, type `TSR` at the DOS prompt and press *Enter.*
3. Run TDMEM to see a memory map of your computer. Note the segment address at which the resident portion of your TSR is loaded. We refer to this segment as *Seg.*
4. Next, you need to determine the amount of symbol table memory you are going to want Turbo Debugger to allocate when you call it up. To do this, note the size of your TSR's symbol table (.TDS) file by doing a DIR command from DOS.

   This size is a lower limit on the amount of symbol table memory you need to allocate when you load Turbo Debugger, since, in addition to the information stored here, Turbo Debugger creates a number of tables, temporary and otherwise, when it loads the symbol table. A useful rule of thumb is that you need to allocate about one and a half times as much symbol table memory as the .TDS file occupies on the disk, though sometimes you might need more and sometimes you can get by with less. Turbo Debugger lets you know if you've allocated too little symbol table memory by displaying the message "Not enough memory to load symbol table" when you do a **File | Symbol** Load (discussed later), so feel free to experiment.
5. Load Turbo Debugger without specifying a file name, allocating symbol table memory as appropriate with the **–sm** command-line option. The **–sm** option takes as an argument the number of kilobytes of symbol table memory to be allocated. For example, if you want to reserve 3K of symbol table memory, enter `TD -sm3` at the DOS prompt. When you load Turbo Debugger, do not specify a file name, since you are debugging something that is already in memory. You should have the .TDS and source files for your TSR available in your default directory, however, so that they can be accessed to supply symbolic information.
6. You could now start debugging your TSR by setting breakpoints, making Turbo Debugger go resident, and performing some action from the DOS command level that would trigger your breakpoint. This opens Turbo Debugger at the appropriate place in your code. However, your debugging task can be simplified by recalling the symbolic information present in your symbol table and source file first.

7. Once Turbo Debugger comes up, clear the sign-on message by pressing *Esc*, then load in your TSR's symbol table with the **F**ile | **S**ymbol Load command, specifying the appropriate symbol table name. If you get a message that there is not enough memory to load your symbol table, exit Turbo Debugger and start it up again from the DOS prompt using a higher value as an argument to **–sm**.
8. The symbol table contains a set of symbols tied to relative memory locations in your code. The symbols in the symbol table are all prefixed by the characters `#FILENAME#`, where *FILENAME* is the name of your TSR source file. For example, if your source file was called TSR.ASM and contained a label *Intr*, the symbol `#TSR#INTR` marks a location in memory.

   The symbols in the symbol table are offset from each other by the correct number of bytes, but the absolute location of the first symbol has not been determined because DOS might have loaded your TSR at a different absolute memory location than the one at which it was assembled. For this reason, you must use a command to explicitly locate the first symbol in memory.
9. Use **F**ile | **T**able Relocate to place the first symbol from the symbol table at the proper location in memory. In this way, the symbolic information present corresponds with your code. To do this, when you are prompted by Turbo Debugger, specify the segment address *Seg* for your TSR that you determined from TDMEM.

   The disassembled statements from memory are synchronized with information from the symbol table. If your source file is present, source statements are printed on the same line as the information from the symbol table.
10. Use the **G**oto command (*Ctrl-G*) to go to the segment of RAM containing your TSR. Do this either by giving the segment address of your TSR, followed by offset 0000H, or by going to a specific symbolic label in your code.

From here on, continue as in the first method, from Step 4 on.

What's a device driver?

Device drivers are collections of routines used by DOS to control low-level I/O functions. Installable device drivers (as opposed to those intrinsic to DOS) are installed by inserting lines such as

```
device = clock.sys
```

in your CONFIG.SYS file. When DOS has to perform an I/O operation involving a single character, it scans through a linked list of device headers looking for a device with the appropriate logical name (for example, COM1). In the case of block device drivers such as disk drives, DOS keeps track of how many block devices have been installed and designates each by a letter, with *A* for the first block device driver installed, *B* for the second, and so on. When you make a reference to drive C, for example, DOS knows to call the third block device driver.

The linked list of device headers contains offsets to the two components of the device driver itself, the *strategy routine* and the *interrupt routine.*

When DOS determines that a given device driver needs to be invoked, it calls the driver twice. The first time the driver is called, DOS talks to the strategy routine and passes it a pointer to a memory buffer called the *request header*. The request header contains information about what DOS wants the device driver to do. The strategy routine simply stores this pointer away for later use. On the second call to the device driver, DOS invokes the interrupt routine, which does the actual work specified by DOS in the request header, such as transferring characters in from a disk.

The request header specifies what the device driver is to do through a byte in the request header called a *command code*. This specifies one of a predefined set of operations all device drivers must perform. The set of command codes is different for character device drivers than for block device drivers.

The problem with debugging device drivers is that there is no .EXE file to run, since for proper operation, the driver must be installed using a DEVICE = *DRIVER.EXT* command at boot, where *.EXT* = .SYS, .COM or .BIN. This means the device driver to be debugged is already resident in memory before debugging, as it must be for proper operation. Hence the functions to load and relocate symbol tables become very useful, since they can restore symbolic information to the disassembled segment of memory where the device driver is loaded. The **File | Resident** command is also very useful, as we shall see.

Debugging a device driver

Here is how you debug a device driver using TDREMOTE:

1. Compile or assemble the device driver, being sure to incorporate symbolic (debugging) information. Use the TASM **/ZI** or TCC **–v** command-line option, for example.
2. Link the device driver using the **/v** option to incorporate symbolic information.
3. Type `TDSTRIP -s -c` *`FILENAME`*, where *FILENAME* is the name of the device you're debugging, to move the symbolic information from the .EXE file into a separate .TDS file, and to transform the existing .EXE file into a .COM file.

   ```
   TDSTRIP -s -c FILENAME
   ```

   where *FILENAME* is the name of the device driver you're debugging. Copy the .COM file to the remote system.
4. Modify your CONFIG.SYS file on the remote system by adding the line

   ```
   device = FILENAME.COM
   ```
5. Make sure *FILENAME* includes the correctpath to find the device driver.
6. Reboot your remote system to load the device driver.
7. Run TDDEV to tell you the location in memory on the remote system where DOS has loaded your device driver. Note the address where your device driver is loaded. We refer to the segment portion of this address as *Seg*.
8. Next you need to determine the amount of symbol table memory you will need Turbo Debugger to allocate when you call it up. To do this, note the size of your device driver's symbol table (.TDS) file by doing a DIR command from DOS.

   This size is a lower limit on the amount of symbol table memory you will need to allocate when you load Turbo Debugger, since in addition to the information stored here, Turbo Debugger creates a number of temporary and other tables when loading the symbol table. A useful rule of thumb is that you need to allocate about one and a half times as much symbol table memory as the .TDS file occupies on disk, though sometimes you need more, and sometimes you can get by with less. Turbo Debugger lets you know if you've allocated too little symbol table memory by displaying the message "Not enough memory to load symbol table" when you do a **File | Symbol Load** (discussed later), so feel free to experiment.

9. Load TDREMOTE on the remote system.
10. Load Turbo Debugger (using the **–r** option and the **–rp** and **–rs** options as needed) *without specifying a file name,* allocating symbol table memory as appropriate by using the **–sm** command line switch. The **–sm** switch takes as an argument the number of kilobytes of symbol table memory to be allocated. For example, if you wish to reserve 3K of symbol table memory, type `TD -sm3` at the DOS prompt. When you load Turbo Debugger, you do not specify a file name because you are debugging something that is already in memory. You should have the .TDS and source files for your device driver available in your default directory, however, so that they can be accessed to supply symbolic information.
11. You could now start debugging your device driver by setting breakpoints, making Turbo Debugger go resident, and performing some action from the DOS command level on the remote system which would trigger your breakpoint. This would open Turbo Debugger at the appropriate place in your code. However, your debugging task can be simplified by recalling the symbolic information present in your symbol table and source file first.
12. Once Turbo Debugger comes up, clear the sign-on message by pressing *Esc,* then load in your device driver's symbol table using the **File | Symbol Load** command, specifying the appropriate symbol table name. If you get a message that there is not enough memory to load your symbol table, exit Turbo Debugger and start it up again from the DOS prompt using a higher value as an argument to **–sm**.
13. The symbol table contains a set of symbols tied to relative memory locations in your code. The symbols in the symbol table are all prefixed by the characters `#FILENAME#`, where *FILENAME* is the name of your device driver source file. For example, if your source file was called DRIVER.ASM and contained a label *Intr,* the symbol `#DRIVER#INTR` marks a location in memory.

    The symbols in the symbol table are offset from each other by the correct number of bytes, but the absolute location of the first symbol is not determined, since DOS may load your device driver at a different absolute memory location than the one at which it was assembled. For this reason, you must use a command to explicitly locate the first symbol in memory.

14. Use the **File | Table Relocate** command to place the first symbol from the symbol table at the proper location in memory. In this way, the symbolic information present will correspond with your code. To do this, specify the segment address *Seg* for your device driver which you determined in Step 6.

    The disassembled statements from memory are synchronized with information from the symbol table. If your source file is present, source statements will be printed on the same line as the information from the symbol table.

15. Set any breakpoints in your code.
16. Choose the **File | Resident** command to make Turbo Debugger itself go resident. This has nothing to do with making your device driver memory resident; it goes resident at boot on the remote system as a result of the device command in CONFIG.SYS. The only reason you are making Turbo Debugger go resident is so you can go back to DOS and do whatever is necessary to invoke your device driver.
17. When you are back to the DOS command line on the remote system, do whatever is necessary to activate your device driver. For example, send information to whatever device it controls.
18. When your program hits the breakpoint, Turbo Debugger comes back up with your device driver displayed at the appropriate point, and you can begin debugging your code. (You can also re-enter Turbo Debugger while DOS is running, by pressing *Ctrl-Break*.)

Terminating the debugging session

To terminate a debugging session, get out of Turbo Debugger in the usual way, by choosing the **File | Quit** command or pressing *Alt-X*. If you're debugging a TSR, it will be unloaded automatically.

A P P E N D I X

A

Summary of command-line options

When you start up Turbo Debugger from the DOS command line, you can at the same time configure it using certain options. Here's the general format to use:

```
td [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 A.1
Turbo Debugger command-line options

Option	What it means
–c*filename*	Startup configuration file
–do	Other display
–dp	Page flipping
–ds	Swap user screen contents
–h, –?	Display help screen listing all the command-line options
–i	Process ID switching
–k	Enable keystroke recording
–l	Assembler startup
–m*N*	Set heap size (K)
–p	Enable mouse
–r	Debug on remote system; COM1, fast
–rp*N*	COM port for remote link
–rs*N*	Link speed: 1=slow, 2=med, 3=fast
–sc	No case-checking
–sd*directory*	Source file directory
–sm*N*	Set symbol table memory size (K)
–vg	Complete graphics save
–vn	43/50 line display not allowed
–vp	EGA palette save
–y*N*	Set overlay pool size (K)
–ye*N*	Set EMS overlay area size to *N* 16K pages

A P P E N D I X

B

Technical notes

This appendix is for advanced users who want to understand some of the technical details that underlie the operation of Turbo Debugger. Don't be put off if this chapter appears to have been written in Greek; you don't have to understand the issues presented here in order to become a productive and successful Turbo Debugger user.

Some of the information in this chapter will let you understand how Turbo Debugger interacts with DOS, the hardware, and your program. This can help you understand how your program's behavior might differ while running under Turbo Debugger.

You will also learn why you can crash the system without too much effort, and, even better, how to avoid it.

Changed load address and free memory

When Turbo Debugger loads your program, it is placed after the debugger in memory. This has two important results: Your program loads at a higher segment address, and it has less free memory available. By loading at a different address, some bugs that are the result of accessing memory outside your program may appear or disappear. By changing the amount of free memory, bugs in your memory allocation or use may be hard to duplicate.

If you're using a 386-based computer, you can use the TD386 virtual debugging program to eliminate those problems. See Chapter 15 for information on virtual debugging.

Crashing the system

Since the debugger can read and write memory at any address in your system, you can inadvertently cause a crash by modifying certain memory locations outside your program, such as some inside DOS, or the interrupt table starting at memory address location zero.

As an example, changing the hardware clock interrupt vector at location 0000h:0040h is almost certain to cause a problem.

Tracing through DOS and process ID switching

Turbo Debugger keeps track of the process that is running (either itself or your program) so that it can open and close files without interfering with your program's file handles. This switching is done by using a DOS function call. The switch occurs each time your program is started from Turbo Debugger, and each time the debugger is re-entered from your program. Since DOS is not re-entrant, you can get into trouble by setting breakpoints or tracing inside DOS.

You should use the **–i-** command-line option to disable process ID switching if you want to poke around inside DOS. However, your program will then share Turbo Debugger's file handles, which may cause either your program or the debugger to run out of them.

Using the 8087/80287 math coprocessor and emulator

Turbo Debugger uses neither the math coprocessor nor the software emulator, leaving them both free to be used by your program. You shouldn't experience any difference between using a standalone floating-point program and running it under Turbo Debugger.

Interrupts used by Turbo Debugger

Turbo Debugger intercepts several interrupt vectors in order to debug your program. The following descriptions let you determine if there may be interactions between your program and Turbo Debugger.

Interrupt 1/Interrupt 3
Turbo Debugger uses these interrupts to process breakpoints and instruction single-stepping. If these interrupts are modified by your program, Turbo Debugger may not be able to regain control at the next breakpoint. Normal applications never use these interrupts because they are reserved for programs such as debuggers that must control the execution of other programs.

Interrupt 2
Many hardware debuggers use this interrupt to signal that a match condition has occurred. If your program takes over this interrupt, these boards and their supporting device drivers may not work properly. If you must take over this interrupt, chain on to the previous owner of it if you do not want to service the interrupt.

Interrupt 9
This is the keyboard hardware interrupt, which is used for tracking key presses and release codes. Turbo Debugger chains into this interrupt when the user program is running, so that it can regain control of a program stuck in a loop. Turbo Debugger reinstalls this vector each time your program is restarted, thereby allowing a program that modifies this interrupt to keep working correctly.

Debugging using INT 3 and INT 1

If you want to debug a program that uses these interrupts, the version of the program you are debugging should only load these interrupt vectors when it absolutely must, and restore the old contents as soon as it is done using them. This technique minimizes the amount of code that cannot be debugged. While your program has these vectors loaded, you cannot use Turbo Debugger to step through your code.

Display-saving and mode-switching

Turbo Debugger usually attempts to save and restore your program's display mode whenever it runs a piece of your program. If you only use the standard ROM BIOS calls to change the display mode, all will be well. If you directly manipulate the display controller registers, Turbo Debugger may disturb those settings.

Memory consumption

When you first start Turbo Debugger, DOS loads it into the first free memory above DOS and any resident programs. Then, Turbo Debugger allocates a working stack and heap above its program code. Your program's symbol table comes next in memory, followed by the actual program that you want to debug.

When you exit back to DOS, Turbo Debugger frees the memory used by the symbol table and the program being debugged. If your program has allocated any memory blocks with the DOS memory allocate function (48), Turbo Debugger frees that memory as well.

EMS support

If your system has an expanded memory specification (EMS) board, Turbo Debugger will use it to store the symbol table for your program being debugged. This leaves more main memory free for your program. Turbo Debugger saves and restores the state of the EMS driver, letting you debug programs that use EMS memory.

If your program must use all of EMS memory, or if you experience interaction problems between your program and Turbo Debugger with both using EMS memory, you can disable EMS symbol table use by Turbo Debugger. Use the TDINST installation utility to do this or specify **–ye***0* to disable overlay caching in EMS.

Interrupt vector saving and restoring

Turbo Debugger maintains three separate copies of the first 48 interrupt vectors in low memory (00 through 2F).

When Turbo Debugger first starts from the DOS command line, a copy is made of the vectors. These vectors are restored when you return back to DOS by using the **File | Quit** (or *Alt-X*) command. These vectors are also restored if you use the **File | DOS** Shell command to enter a DOS command while debugging a program.

The second set of vectors are Turbo Debugger's vectors. These are in effect whenever Turbo Debugger is running and onscreen. They are restored every time Turbo Debugger regains control after running your program.

The third set of vectors are for the program you're debugging. These vectors are restored every time you run or step your program, and are saved every time your program stops and Turbo Debugger regains control. This lets you debug programs that change interrupt vectors, and at the same time allows Turbo Debugger to use its own version of those same interrupts.

A P P E N D I X

C

Inline assembler keywords

This appendix lists the instruction mnemonics and other special symbols that you use when entering instructions with the inline assembler. The keywords presented here are the same as those used by Turbo Assembler and MASM.

Table C.1
8086/80186/80286 instruction mnemonics

AAA	INC	LIDT**	REPNZ
AAD	INSB*	LLDT**	REPZ
AAM	INSW*	LMSW**	RET
AAS	INT	LOCK	REFT
ADC	INTO	LODSB	ROL
ADD	IRET	LODSW	ROR
AND	JB	LOOP	SAHF
ARPL**	JBE	LOOPNZ	SAR
BOUND*	JCXZ	LOOPZ	SBB
CALL	JE	LSL**	SCASB
CLC	JL	LTR**	SCASW
CLD	JLE	MOV	SGDT**
CLI	JMP	MOVSB	SHL
CLTS**	JNB	MOVSW	SHR
CMC	JNBE	MUL	SLDT**
CMP	JNE	NEG	SMSW**
CMPSB	JNLE	NOP	STC
CMPSW	JNO	NOT	STD
CWD	JNP	OR	STI
DAA	JO	OUT	STOSB
DAS	JP	OUTSB	STOSW
DEC	JS	OUTSW*	STR**
DIV	LAHF	POP	SUB
ENTER*	LAR**	POPA*	TEST
ESC	LDS	POPF	WAIT
HLT	LEA	PUSH	VERR**
IDIV	LEAVE*	PUSHA*	VERW**
IMUL	LES	PUSHF	XCHG
IN	LGDT**	RCL	XLAT
			XOR

* Available only when running on the 186 and 286 processor
** Available only when running on the 286 processor

Table C.2
80386 instruction mnemonics

BSF	LSS	SETG	SETS
BSR	MOVSX	SETL	SHLD
BT	MOVZX	SETLE	SHRD
BTC	POPAD	SETNB	CMPSD
BTR	POPFD	SETNE	STOSD
BTS	PUSHAD	SETNL	LODSD
CDQ	PUSHFD	SETNO	MOVSD
CWDE	SETA	SETNP	SCASD
IRETD	SETB	SETNS	INSD
LFS	SETBE	SETO	OUTSD
LGS	SETE	SETP	JECXZ

Table C.3
80486 instruction mnemonics

BSWAP	**INVLPG**
CMPXCHG	**WBPINVD**
INVD	**XADD**

Table C.4
80386 registers

EAX	**EDI**
EBX	**EBP**
ECX	**ESP**
EDX	**FS**
ESI	**GS**

Table C.5
CPU registers

Byte registers	**AH, AL, BH, BL, CH, CL, DH, DL**
Word registers	**AX, BX, CX, DX, SI, DI, SP, BP, FLAGS**
Segment registers	**CS, DS, ES, SS**
Floating registers	**ST, ST(0), ST(1), ST(2), ST(3), ST(4), ST(5), ST(6), ST(7)**

Table C.6
Special keywords

WORD PTR	**TBYTE PTR**
BYTE PTR	**NEAR**
DWORD PTR	**FAR**
QWORD PTR	**SHORT**

Table 0.1
8087/80287 numeric coprocessor instruction mnemonics

FABS	FIADD	FLDL2E	FST
FADD	FICOM	FLDL2T	FSTCW
FADDP	FICOMP	FLDPI	FSTENV
FBLD	FIDIV	FLDZ	FSTP
FBSTP	FIDIVR	FLD1	FSTSW*
FCHS	FILD	FMUL	FSUB
FCLEX	FIMUL	FMULP	FSUBP
FCOM	FINCSTP	FNOP	FSUBR
FCOMP	FINIT	FNSTS**	FSUBRP
FCOMPP	FIST	FPATAN	FTST
FDECSTP	FISTP	FPREM	FWAIT
FDISI	FISUB	FPTAN	FXAM
FDIV	FISUBR	FRNDINT	FXCH
FDIVP	FLD	FRSTOR	FXTRACT
FDIVR	FLDCW	FSAVE	FYL2X
FDIVRP	FLDENV	FSCALE	FYL2XP1
FENI	FLDLG2	FSETPM*	F2XM1
FFREE	FLDLN2	FSQRT	

* Available only when running on the 287 numeric coprocessor.
** On the 80287, the **fstsw** and **fnstsw** instructions can use the AX register as an operand, as well as the normal memory operand.

Table 0.2
80387 instruction mnemonics

FCOS	FUCOM
FSIN	FUCOMP
FPREM1	FUCOMPP
FSINCOS	

A P P E N D I X

D

Customizing Turbo Debugger

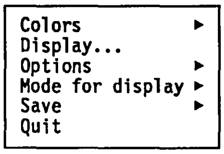
Turbo Debugger is ready to run as soon as you make working copies of the files on the distribution disk. However, you can change many of the default settings by running the customization program called TDINST. You also can change some of the options using command-line options when you start Turbo Debugger from DOS. If you find yourself frequently specifying the same command-line options over and over, you can make those options permanent by running the customization program.

The customization program lets you set the following items:

- Window, dialog box, and menu colors
- Display parameters: screen swapping mode, integer display format, beginning display (source or assembler code), screen lines, tab column width, maximum tiled Watches size, fast screen update, 43-/50-line mode, full graphics saving, User screen updating, and log list length
- Your editor startup command and directories to search for source files and the Turbo Debugger help and configuration files
- User input and prompting parameters: interrupt key, history list length, beep on error, mouse, keystroke recording, and control-key shortcuts

- Source debugging: language options and case sensitivity
- NMI intercept, DOS process ID switching, expanded memory specification (EMS) for symbol table, remote debugging, OS shell swap size, and symbol memory size
- Display mode

Running TDINST



To run the customization program, enter `TDINST` at the DOS prompt. As soon as TDINST comes up, it displays its main menu. You can either press the highlighted first letter of a menu option or use the ↑ and ↓ keys to move to the item you want and then press *Enter*. For instance, press *D* to change the display settings. Use this same technique for choosing from the other menus in the installation utility. To return to a previous menu, press *Esc*. You may have to press *Esc* several times to get back to the main menu.

Setting the screen colors

Choose **C**olors from the main menu to bring up the **C**olors menu. It offers you two choices: **C**ustomize and **D**efault Color Set.

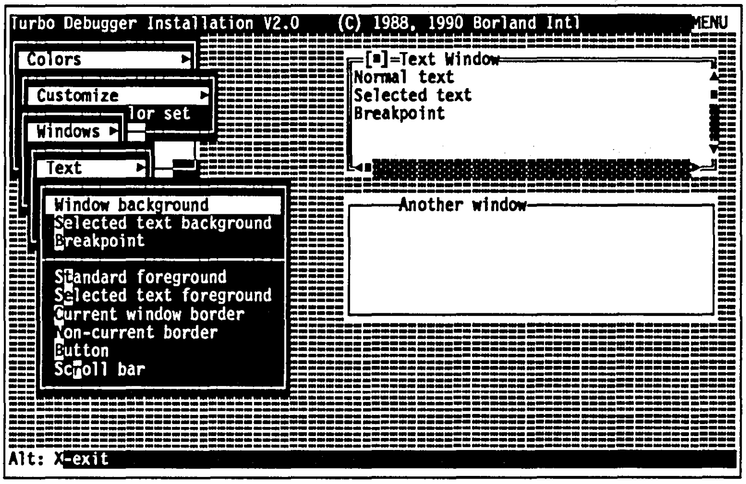
Customizing screen colors

If you choose **C**ustomize, a third menu appears, with options for customizing windows, dialog boxes, menus, and screens.

Windows

To customize windows, choose the **W**indows command. This opens a fourth menu, from which you can choose the kind of window you want to customize: **T**ext, **D**ata, **L**ow Level (for example, the CPU window), and **O**ther (for example, the Breakpoints window). Choosing one of these options brings up yet another menu listing the window elements, together with a pair of sample windows (one active, one inactive) in which you can test various color combinations. The screen looks like this:

Figure D.1
Customizing colors for windows


```
Turbo Debugger Installation V2.0     (C) 1988, 1990 Borland Intl        MENU
 Colors          ►        =[■]=Text Window
  Customize       ►        Normal text
           lor set         Selected text
  Windows ►                Breakpoint
   Text      ►
    Window background              Another window
    Selected text background
    Breakpoint
    ─────────────────────
    Standard foreground
    Selected text foreground
    Current window border
    Non-current border
    Button
    Scroll bar
Alt: X-exit
```

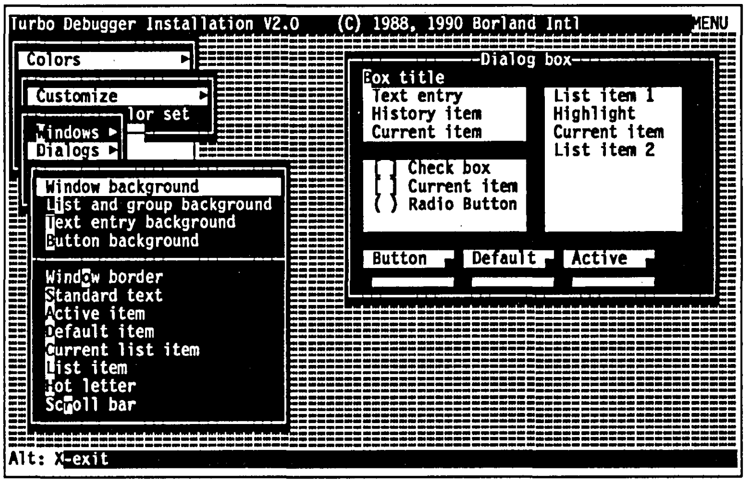
When you select an item you want to change, a palette box pops up over the menu. Use the arrow keys to move around in the palette box. As you move the selection box through the various color choices, the window element whose color you are changing is updated to show the current selection. When you find the color you like, press *Enter* to accept it.

Turbo Debugger maintains three color tables: one for color, one for black and white, and one for monochrome. You can only change one set of colors at a time, based on your current video mode and display hardware. So, if you are running on a color display and want to adjust the black-and-white table, first set your video mode to black and white by typing `MODE BW80` at the DOS prompt, and then run TDINST.

Dialog boxes

If you choose **D**ialogs from the **C**ustomize menu, a menu appears listing dialog box and menu elements, with a sample dialog box for you to experiment with.

The screen looks like this:



```
Turbo Debugger Installation V2.0    (C) 1988, 1990 Borland Intl        MENU
 Colors        ►                   ------Dialog box------
  Customize     ►                  Box title
          lor set                   Text entry           List item 1
  Windows ►                         History item         Highlight
  Dialogs ►                         Current item         Current item
                                                         List item 2
   Window background                [ ] Check box
   List and group background        [ ] Current item
   Text entry background            ( ) Radio Button
   Button background
                                    Button    Default    Active
   Window border
   Standard text
   Active item
   Default item
   Current list item
   List item
   Hot letter
   Scroll bar
Alt: X-exit
```

Figure D.2
Customizing colors for dialog

As with the **W**indows menu, choosing an item from the current menu opens a palette from which you can choose the color for that item.

Menus

```
Menu background
Standard item
Active item
Hot letter
```

If you choose **M**enus from the **C**ustomize menu, a menu of menu options opens, along with a sample menu. Choosing an item from the menu causes the usual palette to appear.

Screen

```
Pattern for background ►

Pattern background
Pattern foreground
Window move background

Window move foreground
```

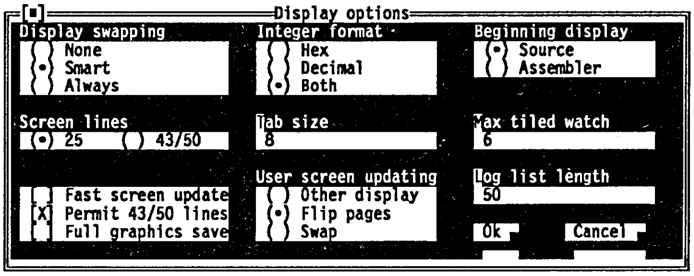
Choosing **S**creen from the **C**ustomize menu opens a menu from which you can access another menu with screen patterns and palettes for screen elements, as well as a sample screen background on which to test them.

The default colors

If you choose **D**efault Color Set from the **C**olors menu, an active text window and an inactive window appear onscreen, so you can see what the default colors for their elements are.

Setting Turbo Debugger display parameters

Choose **D**isplay from the main menu to bring up the Display Options dialog box.



```
[■]                    Display options
Display swapping        Integer format          Beginning display
 ( ) None                ( ) Hex                 (•) Source
 (•) Smart               ( ) Decimal             ( ) Assembler
 ( ) Always              (•) Both

Screen lines            Tab size                Max tiled watch
 (•) 25    ( ) 43/50     8                       6

                        User screen updating    Log list length
 [ ] Fast screen update  ( ) Other display       50
 [X] Permit 43/50 lines  (•) Flip pages
 [ ] Full graphics save  ( ) Swap                Ok        Cancel
```

Figure D.3
The Display Options dialog box

These display options include some you can set from the DOS command line when you start up Turbo Debugger, as well as some you can set only with TDINST. See page 291 for a table of Turbo Debugger command-line options and corresponding TDINST settings.

Display Swapping

You use the Display Swapping radio buttons to control how Turbo Debugger switches between its own display and the output of the program you're debugging. You can toggle between the following settings:

None	Don't swap between the two screens. Use this option if you're debugging a program that does not output to the User screen.
Smart	Swap to the User screen only when display output may occur. Turbo Debugger swaps the screens any time that you step over a routine, or if you execute any instruction or source line that appears to read or write video memory. This is the default option.
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, since Turbo Debugger's screen is replaced for a short time with the User screen.

Integer Format

The Integer Format radio buttons let you set how integers are displayed. You can toggle between the following options:

Hex	Chooses hexadecimal number display.
Decimal	Chooses decimal number display.
Both	Displays both hexadecimal and decimal.

Beginning Display

The Beginning Display radio buttons determines the language in which your program is displayed when Turbo Debugger starts. They have the following settings:

Assembler	Assembler Startup: None of your program is executed, and a CPU window shows the first instruction in your program.
Source	Source startup: Your program's compiler beginning code runs, and you start in a Module window, where your source code begins.

Screen Lines

Use these radio buttons to toggle whether Turbo Debugger should start up with a display screen of 25 lines or a display screen of 43 or 50 lines.

☞ Only the EGA and VGA can display more than 25 lines.

Tab Size

In this input box, you can set the number of columns between tab stops in a text or source file display. You are prompted for the number of columns (a number from 1 to 32); the default is 8.

Maximum Tiled Watch

This input box sets the number of lines that the Watches window can expand to when it's in Tiled mode. You are prompted for the number of lines (1 to 20).

Fast Screen Update

The Fast Screen Update check box lets you toggle whether your displays will be updated quickly. Toggle this option off if you get "snow" on your display with fast updating enabled. You need to disable this option only if the "snow" annoys you. (Some people prefer the snowy screen because it gets updated more quickly.)

Permit 43/50 Lines

Turning this check box on allows big (43-/50-line) display modes. If you turn it off, you save approximately 8K, since the large screen modes need more window buffer space in Turbo Debugger. This may be helpful if you are debugging a very large program that needs as much memory as possible to execute in. When the option is disabled, you will not be able to switch the display into 43-/50-line mode even if your system is capable of handling it.

Full Graphics Saving

Turning this check box on causes the entire graphics display buffer to be saved whenever there is a switch between the Turbo Debugger screen and the User screen. If you turn it off, you can save approximately 8K of memory. This is helpful if you are debugging a very large program that needs as much memory as possible to execute. Generally the only drawback to disabling this option is a small number of corrupted locations on the User screen in graphics mode that don't usually interfere with debugging.

User Screen Updating

The User Screen Updating radio buttons set how the User screen is updated when Turbo Debugger switches between its screen and your program's User screen. There are three settings:

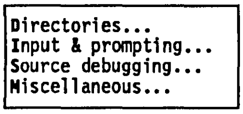
Flip Pages Puts Turbo Debugger's screen on a separate display page. This option works only if your display adapter has multiple display pages, like a CGA, EGA, or VGA. You can't use this option on a monochrome display. This option works for the majority of debugging situations; it is fast and

	disturbs only the operation of programs that use multiple display pages, such as graphics programs.
Swap	Uses a single display adapter and display page, and swaps the contents of the User and Turbo Debugger screens in software. This is the slowest method of display swapping, but it is the most protective and least disruptive. If you are debugging a program that uses multiple display pages, like a graphics program, use this option. Also use the Swap option if you shell to DOS and run other utilities or if you are using a TSR (such as SideKick Plus) and want to keep the current Turbo Debugger screen as well.
Other Display	Runs Turbo Debugger on the other display in your system. If you have both a color and monochrome display adapter, this option lets you view your program's screen on one display and Turbo Debuggers on the other.

Log List Length

Use this input box to set how many previous entries are saved in the log file. The maximum number is 200; the minimum is 4.

Turbo Debugger options


```
Directories...
Input & prompting...
Source debugging...
Miscellaneous...
```

The **Options** command in the main menu opens a menu of options, which in turn open dialog boxes for you.

Directories...

This dialog box contains input boxes in which you can enter:

Editor program name	Specifies the DOS command that starts your editor. This lets Turbo Debugger start up your favorite editor when you are debugging and want to change something in a file. Turbo Debugger adds to the end of this command the name of the

file that it wants to edit, separated by a space.

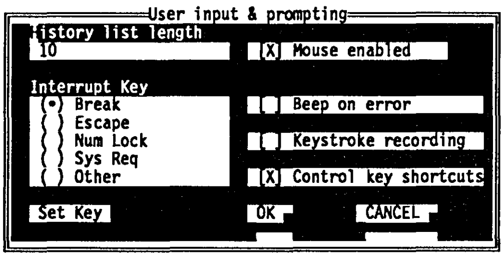
Source directories Sets the list of directories Turbo Debugger searches for source files.

Turbo directory Sets the directory that Turbo Debugger will look in for its help and configuration files.

Input and Prompting...

This dialog box lets you set options that control how you input information to Turbo Debugger, and how Turbo Debugger prompts you for information:

Figure D.4
The User Input and Prompting dialog box



```
User input & prompting
History list length
 10                         [X] Mouse enabled

Interrupt Key
 (•) Break                  [ ] Beep on error
 ( ) Escape
 ( ) Num Lock               [ ] Keystroke recording
 ( ) Sys Req
 ( ) Other                  [X] Control key shortcuts

 Set Key                    OK        CANCEL
```

History List Length This input box lets you specify how many earlier entries are to be saved in an history list input box.

Interrupt Key These radio buttons let you assign a default interrupt key.

Set Key If you choose Other, press the Set Key button to choose the actual interrupt key. You are prompted for the key to use.

Mouse Enabled This check box controls whether Turbo Debugger defaults to mouse support.

Beep on Error By default, Turbo Debugger gives a warning beep when you press an invalid key or do something that generates an error message. The Beep on Error check box lets you change this default.

Keystroke Recording — This check box determines whether the Execution History window defaults to automatic keystroke recording.

Control Key Shortcuts — This check box enables or disables the control-key shortcuts. When control-key shortcuts are enabled, you can invoke any local menu command directly by pressing the *Ctrl* key in combination with the first letter of the menu item. However, in that case, you can't use those control keys as WordStar-style cursor-movement commands.

Source Debugging...

The Source Debugging dialog box lets you specify what language Turbo Debugger will use for evaluating expressions, and enables and disables case sensitivity.

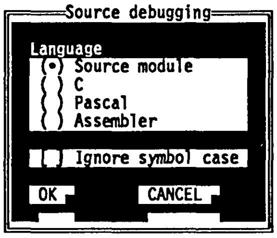


Figure D.5
The Source Debugging dialog box

```
Source debugging
Language
(•) Source module
( ) C
( ) Pascal
( ) Assembler
[ ] Ignore symbol case
OK    CANCEL
```

Language — The Language radio buttons toggle the language Turbo Debugger uses for evaluating expressions:

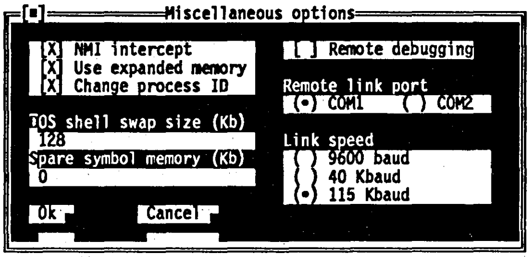
Source Module	Choose what language to use based on the languages of the current source module.
C	Always use C expressions, no matter what language the current module was written in.
Pascal	Always use Pascal expressions, no matter language the current module was written in.
Assembler	Always use assembler expressions, no matter what language the current module was written in.

Ignore Symbol Case — If this check box is turned on, Turbo Debugger defaults to treating uppercase and lowercase the same. If it is off, case sensitivity is in effect.

Miscellaneous Options...

The Miscellaneous Options dialog box contains options controlling NMI interrupts, EMS memory, use of process IDs DOS shell swapping, symbol table size, and remote debugging.

Figure D.6
The Miscellaneous Options dialog box



```
[■]═══════════Miscellaneous options═══════════
[X] NMI intercept               [ ] Remote debugging
[X] Use expanded memory
[X] Change process ID           Remote link port
                                (•) COM1    ( ) COM2
DOS shell swap size (Kb)
128                             Link speed
Spare symbol memory (Kb)        ( ) 9600 baud
0                               ( ) 40 Kbaud
                                (•) 115 Kbaud
```

NMI Intercept

If your computer is a Tandy 1000A, IBM PC Convertible, or NEC MultiSpeed, or if Turbo Debugger hangs loading your system, run TDINST and turn off the NMI Intercept check box. Some computers use the NMI (nonmaskable interrupt) in ways that conflict with Turbo Debugger, so you must disable Turbo Debugger's use of this interrupt in order to run the program.

Use Expanded Memory

Use this check box to toggle whether Turbo Debugger uses EMS memory for symbol tables. You can enable this option even if your program uses EMS as well.

Change Process ID

Use this check box to control whether Turbo Debugger uses process ID switching.

Warning! Do not turn this check box off unless you are tracing through DOS and have a good understanding of the technical issues discussed in Appendix B.

DOS Shell Swap Size

Determines how much of the user program is swapped to disk when you shell to DOS; if you enter 0, the whole program is swapped.

Spare Symbol Memory

This input box lets you specify the amount of memory set aside for manually loaded symbol tables.

Remote Debugging This check box lets you toggle between enabling and disabling the remote link.

Warning! Usually you won't want to turn this check box on, since that will mean that Turbo Debugger will start up every time using the remote link.

Remote Link Port The Remote Link Port radio buttons let you choose between using the COM1 or COM2 serial port for the remote link.

Link Speed The Link Speed radio buttons let you choose one of the three speeds that are available for the remote link: 9600 baud, 40,000 baud, or 115,000 baud.

Setting the mode for display

```
Default
Color
Black and white
Monochrome
LCP
```

Choosing Mode for Display from the main menu opens a menu from which you can select the display mode for your system.

Default

Turbo Debugger detects the kind of graphics adapter on your system and selects the display mode appropriate for it.

Color

If you have an EGA, VGA, CGA, MCGA, or 8514 graphics adapter and choose this as your default, the display will be in color.

Black and White

If you have an EGA, VGA, CGA, MCGA, or 8514 graphics adapter and choose this as your default, the display will be in black and white.

Monochrome

Choose this if you are using a color monitor with a Hercules or monochrome text-only adapter.

LCD

Choose this if you have an LCD monitor.

Command-line options and installation equivalents

Some of the options described in the previous section can be overridden when you start Turbo Debugger from DOS. The following table shows the correspondence between Turbo Debugger command-line options and the TDINST program command that permanently sets that option.

Table D.1
Turbo Debugger command-line options

Option	TDINST menu path	Dialog box and option
	Display	Display Options
-do		`(•) Other Display`
-dp		`(•) Flip Pages`
-ds		`(•) Swap`
	Options | Miscellaneous	Miscellaneous Options
-i		`[X] Change Process ID`
-i-		`[ ] Change Process ID`
	Options | Input and Prompting	User Input and Prompting
-k		`[X] Keystroke Recording`
-k-		`[ ] Keystroke Recording`
	Display	Display Options
-l		`(•) Assembler`
-l-		`(•) Source`
	Options | Input and Prompting	User Input and Prompting
-p		`[X] Mouse Enabled`
-p-		`[ ] Mouse Enabled`
	Options | Miscellaneous	Miscellaneous Options
-r		`[X] Remote Debugging`
-r-		`[ ] Remote Debugging`
	Options | Miscellaneous	Miscellaneous Options
-rp1		`(•) COM1`
-rp2		`(•) COM2`
	Options | Miscellaneous	Miscellaneous Options
-rs1		`(•) 9600 Baud`
-rs2		`(•) 40 KBaud`
-rs3		`(•) 115 KBaud`
	Options | Source Debugging	Source Debugging
-sc		`[X] Ignore Symbol Case`
-sc-		`[ ] Ignore Symbol Case`
	Options | Directories	Directories
-sd		`Source Directories`

Table D.1: Turbo Debugger command-line options (continued)

Option	TDINST menu path	Dialog box and option
	Options | Miscellaneous	Miscellaneous Options
–sm		`Spare Symbol Memory`
	Display	Display Options
–vn		`[ ] Permit 43/50 Lines`
–vn–		`[X] Permit 43/50 Lines`

For a list of all the command-line options available for TDINST.EXE, enter the program name followed by `-h`:

When you're through...

Saving changes

```
Save configuration file...
Modify td.exe
```

When you have all your Turbo Debugger options set the way you want, choose **S**ave from the main menu to determine how you want them saved.

Save Configuration File

If you choose **S**ave Configuration File, a dialog box opens, initialized to the default configuration file TDCONFIG.TD. You can accept this name by pressing *Enter*, or you can type a new configuration file name. If you specify a different file name, you can load that configuration by using the **–c** command-line option when you start Turbo Debugger. For example,

```
td -cmycfg myprog
```

You can also use the Turbo Debugger **O**ptions | **R**estore Configuration command to load a configuration once you have started Turbo Debugger.

Modify TD.EXE

If you choose **M**odify TD.EXE, any changes that you have made to the configuration are saved directly into the Turbo Debugger executable program file TD.EXE. The next time you enter Turbo Debugger, those settings will be your defaults.

If at any time, you want to return to the default configuration that Turbo Debugger is shipped with, copy TD.EXE from your master disk onto your working system disk, overwriting the TD.EXE file that you modified.

Exiting TDINST

To get out of TDINST at any time, choose **Q**uit from the main menu.

A P P E N D I X

E

Remote debugging

Turbo Debugger's remote capability is not like that offered by other debuggers. With other debuggers, you merely control the debugger from the remote system; the debugger and the program being debugged are both still on the same system. This can cause problems if the program you are debugging requires more memory than that left after the debugger is loaded. TDREMOTE, supplied as part of the Turbo Debugger package, solves this problem by letting you run Turbo Debugger on one system and the program you are debugging on another system.

In this appendix, we'll look at how to debug very large programs by using a second PC connected to your main PC.

Of course, you're probably wondering, "Why use remote debugging?" As an example, if the program you want to debug won't load under Turbo Debugger, you're a candidate for remote debugging. If you get the message "Not enough memory to load symbol table," or the message "Not enough memory" when you attempt to load a program to debug, you may want to consider remote debugging.

Sometimes, your program will load properly under Turbo Debugger, but there may not be enough memory left for it to operate properly. This is another situation where you may want to use remote debugging.

If you're experiencing memory problems debugging a program and your system has EMS memory, make sure you're using EMS it for symbol tables. Usually, Turbo Debugger does this automatically. You can use the configuration utility (TDINST) to control whether Turbo Debugger uses EMS for symbol tables. You can use TDREMOTE to debug TSRs and device drivers that can't be debugged on a single machine.

Setting up a remote debugging system

In order to use the remote debugging facility, you'll need the following equipment:

- a development system with a serial port
- another PC with a serial port and enough memory and disk space to hold the program you want to debug
- a "null modem" or "printer" cable to connect the two systems

Make sure that the cable you use to connect the two systems is set up properly. You can't use a "straight through" extension-type cable. The cable must, at the very least, swap the transmit and receive data lines. (A good computer store should be able to sell you what you need.)

Once you have procured a suitable cable, use it to connect the two serial ports. This completes the hardware setup required for the remote link.

Remote software installation

Copy the remote debugging driver TDREMOTE.EXE onto the remote system. You must also put on the remote system any files required by the program you are debugging. This includes data input files, configuration files, help files, and so on.

You can put files on the remote system by using floppy disks, or you can use the TDRF remote file transfer utility described in the disk-based documentation for the Turbo Debugger utilities.

You can, if you want, put a copy of the program you want to debug onto the remote system. This is not essential, since Turbo Debugger will send it over the remote link if necessary.

Starting the remote link

When you start the TDREMOTE driver program on the remote system, make sure that your current directory is set where you want it. This is important because TDREMOTE puts the program you are going to debug into the current directory at the time TDREMOTE was started.

Before starting TDREMOTE, determine whether your serial port on the remote system is set up as COM1 or COM2. If your serial port is set up as COM1, start up TDREMOTE by typing

```
TDREMOTE -rp1 -rs3
```

If your serial port is set up as COM2, start up TDREMOTE by typing

```
TDREMOTE -rp2 -rs3
```

Both of these commands start the remote link at its maximum speed (115 Kbaud). This will work with most PCs and cable setups. Later, we'll tell you how to start the link at a slower speed if you experience communication difficulties.

TDREMOTE will sign on with a copyright message and indicate that it is waiting for you to start Turbo Debugger on the other end of the link. If you want to stop and return to DOS, just press *Ctrl-Break.*

Starting Turbo Debugger on the remote link

To start Turbo Debugger using the remote link, add the following options to the command line you use to start Turbo Debugger from DOS:

- For serial port COM1: `-rp1 -rs3`
- For serial port COM2: `-rp2 -rs3`

When the link is successfully started, the message "Turbo Debugger online" appears on the remote system, and the message "TDREMOTE online" appears on the Turbo Debugger screen. This will be quickly replaced with Turbo Debugger's normal window display.

Notice that both Turbo Debugger and TDREMOTE use the same command-line options to set the speed and serial port. Both Turbo

Debugger and TDREMOTE must be set to the same speed (**-rs** option) to work properly.

Turbo Debugger also has the **–r** command-line option, which indicates to start the remote link using the default speed and serial port. Unless you've changed the defaults using TDINST, **–r** specifies COM1 at 115,000 baud (the fastest baud speed.)

Here's a typical Turbo Debugger command line to start the remote link:

```
td -rs3 myprog
```

This begins the link on the default serial port (usually COM1) at the highest link speed (115 Kbaud), and loads the program *myprog* into the remote system if it's not already there.

About loading the program to the remote system

Turbo Debugger is smart about loading the program onto the remote system. It looks at the date and time of the copy of the program on the local system and the remote system. If the local copy is later than the remote copy, it presumes you've recompiled or linked the program and sends it over the link at the highest link speed; this happens at a rate of about 11K/second. This means a typical 60K program will take about 6 seconds to transfer, so don't be alarmed if there's a delay when you want to load a new program.

To indicate that something's happening, the screen on the remote system counts up the bytes of the file as they are transferred.

TDREMOTE command-line options

Here is a complete list of all the command-line options supported by TDREMOTE. You can start an option with either a hyphen (-) or a slash (/).

–? or **–h**	Displays a help screen
–rp1	Port 1 (COM1); default
–rp2	Port 2 (COM2)
–rs1	Slow speed, 9600 baud
–rs2	Medium speed, 40,000 baud
–rs3	High speed, 115,000 baud (default)
–w	Writes options to executable program file

If you start TDREMOTE with no command-line options, it uses the default port and speed built into the executable program file

(COM1 and 115,000 baud), unless you have changed them with the **–w** option.

You can make the TDREMOTE command-line options permanent by writing them back into the TDREMOTE executable program file on disk. Do this by specifying the **–w** command-line option along with the other options that you want to make permanent. You are then prompted for the name of the executable program. You can enter a new executable file name that does not already exist. TDREMOTE will create the new executable file.

Note: For a list of all the command-line options available for TDREMOTE, enter the program name followed by `-h`:

```
TDREMOTE -h
```

If you are running on DOS 3.0 or later, the prompt indicates the path and file name that you executed TDREMOTE from. You can accept this name by pressing *Enter*, or you can enter a new executable file name. The new name must already exist and be a copy of the TDREMOTE program that you have already made.

If you are running on a DOS 2.0, you'll have to supply the full path and file name of the executable program.

Remote debugging sessions

Once you've started TDREMOTE and Turbo Debugger in remote mode, you can debug your program much as if you were doing it on a single system. Turbo Debugger commands work exactly as usual; there is nothing new to learn.

Remember that since the program you are debugging is actually running on the remote system, any screen output or keyboard input to the program happens on the remote system. The **Window | User Screen** command has no effect when you are running on the remote link.

The CPU type of the remote system appears as part of the CPU window title, with the word "REMOTE" before it.

If you want to send files over to the remote system while you are running Turbo Debugger, you can go to DOS using the **File | DOS Shell** command and then use the TDRF utility to perform file maintenance activities on the remote system. You can then return to Turbo Debugger by typing `EXIT` at the DOS prompt and continue debugging your program. TDRF is described in the disk-based documentation for Turbo Debugger utilities.

TDREMOTE messages

Here is a list of the messages you might receive when you're working with TDREMOTE.

***nn* bytes downloaded**
A file is being sent to the remote system. This message shows the progress of the file transfer. At the highest link speed (115,000 baud), transfer speed is about 10K per second.

Can't create file
TDREMOTE can't create a file that needs to be sent to it. This can happen either if the disk is full, or the file name already exists as a directory.

Can't modify exe file
The file name you specified to modify is not a valid copy of the TDREMOTE utility. You can only modify a copy of the TDREMOTE utility with the **–w** option.

Can't open exe file to modify
The file name you specified to be modified can't be opened. You have probably entered an invalid or nonexistent file name.

Download complete
A file has been successfully sent to TDREMOTE.

Download failed, write error on disk
TDREMOTE can't write part of a received file to disk. This usually happens when the disk fills up. You will have to delete some files before the file can be successfully downloaded.

Enter program file name to modify
If you are running on DOS 3.0 or later, the prompt will indicate the path and file name that you executed TDREMOTE from. You can accept this name by pressing *Enter*, or you can enter a new executable file name. The new name must already exist and be a copy of the TDREMOTE program that you have already made.

If you're running DOS 2.0, you will have to supply the full path and file name of the executable program.

Interrupted
You have pressed *Ctrl-Break* while waiting for communications to be established with the other system.

Invalid command-line option
You have given an invalid command-line option when starting TDRF from the DOS command line.

Link broken
The program communicating with TDREMOTE has stopped and returned to DOS.

Link established
A program on the other system has just started to communicate with TDREMOTE.

Loading program "*name*" from disk
Turbo Debugger has told TDREMOTE to load a program from disk into memory in preparation for debugging it.

Program load failed, EXEC failure
DOS could not load the program into memory. This can happen if the program has become corrupted or truncated. You should delete the program file from disk to force Turbo Debugger to send a new copy over the link. If this message happens again after deleting the file, you should relink it on the other system and try again.

Program load failed; not enough memory
The remote system does not have enough free memory to load the program that you want to debug. This won't happen except with very large programs, since TDREMOTE takes only about 15K of memory.

Program load failed; program not found
TDREMOTE could not find the program on its disk. This should never happen because Turbo Debugger downloads the program to the remote system if it can't find it.

Program load successful
TDREMOTE has finished loading the program Turbo Debugger wants to debug.

Reading file "*name*" from Turbo Debugger
A file is being sent to Turbo Debugger.

Unknown request: *message*
TDREMOTE has received an invalid request from the other system. This message should never occur if the link is working properly. If you get this message, check that the link cable is in good working order, and if you still keep getting this error, try reducing the link speed by using the **–rs** command-line option.

Waiting for handshake (press Ctrl-Break to quit)
TDREMOTE has been started and is waiting for a program on the other system to start talking to it. If you want to return to DOS before the other system initiates communication, press the *Ctrl-Break* key combination.

Getting it all to work

Since the remote debugging setup involves two different computers and a cable going between them, there's a chance you'll run into some difficulty getting everything to work together.

If you do experience any problems, first check your cable hook-ups. Next, try running the link at the slowest speed by using the **-rs1** command-line option when starting up both TDREMOTE and Turbo Debugger. If it works okay using **-rs1**, try **-rs2** (the middle speed). Some hardware and cable combinations don't always work properly at the highest speed, so if you can only get it to work at a lower speed, you might want to try a different cable or different computers.

A P P E N D I X

F

Dialog boxes and error messages

Turbo Debugger displays error messages and dialog boxes at the current cursor location. This chapter describes the dialog boxes and error and information messages Turbo Debugger generates.

We tell you how to respond to both dialog boxes and error messages. All the dialog boxes and error messages (including the startup fatal error messages) are listed in alphabetical order, with a description provided for each one.

Dialog boxes

Turbo Debugger 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 hard 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 unwritable 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 the number of bytes you want to set, 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 parameters 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 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 evaluate
Enter an expression whose value you want to know. The value and type of the result will be displayed in an error-type window, which disappears once the next keystroke is pressed.

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. Appendix C has a condensed listing of all assembler keywords, and Chapter 11 discusses the assembler 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 **C**lose 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
Enter a single memory address. You can use a symbol name or a complete expression.

Enter memory address, count
Enter a memory address, followed by an optional comma and the number of items. You can use a symbol name or a complete expression.

Enter name of configuration file
Enter the name of a configuration file to read or write. If you are reading from a configuration file, you can enter a wildcard mask and get a list of matching files.

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 relocation segment value
Enter an expression in the current language. This value will be used to set the base segment address of a symbol table that you loaded with the **File | Symbol** Load command. The expression that you enter should evaluate to the segment number of the start of the code for which the symbol table applies.

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-style 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 **A**ssemble local menu command, or enter a byte list as you would for a **S**earch 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 file-name extension.

Enter tab column spacing
Enter a number between 1 and 32 that specifies how far apart tab columns will be when Turbo Debugger displays files in a File or Module window.

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 the **View | File** menu command.

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 symbol
Pick a symbol 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 window
Pick a window from the list of active window titles.

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. This macro will be recorded on disk along with any other keystroke macros.

Press key to delete macro from
Press the key for the macro that you want to delete. The key will then be returned to its original pre-macro function.

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.

Program out of date on remote; send over link?
You are running Turbo Debugger over the remote link, and the program you want to debug is either not on the remote system or it is older than the version on the main system. If you respond *Y*, the new program will be sent over the remote link. If you respond *N*, the load command will be aborted. If you are running at the slowest remote speed, you may want to copy the program to the remote system manually by using a floppy disk. At the highest link speed, the data transfer rate is at least as fast as using a floppy disk.

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

Turbo Debugger 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.

Error messages are normally accompanied by a beep. You can turn off the beep in the customization program, TDINST.

Fatal errors

All fatal errors cause Turbo Debugger to quit and return to DOS. Some fatal errors are the result of trying to start Turbo Debugger from DOS. A few others occur if something unrecoverable happens while you are using the debugger. In either case, after having solved the problem, your only remedy is to restart Turbo Debugger from the DOS prompt.

Bad configuration file
The configuration file is either corrupted or not a Turbo Debugger configuration file.

Could not create dummy PSP segment
When starting the TD386 virtual debugger with no program to load, the dummy program could not be created. Try starting TD386 with a program to debug.

Fatal EMS Error
The EMS memory driver returned an unrecoverable error indication. Either your EMS hardware is malfunctioning, or the software driver has become corrupted. Reboot your system and try again. If the problem persists, it's probably a problem with your EMS hardware.

Invalid switch: __
You supplied an invalid option switch on the DOS command line. Appendix A has an abbreviated list of all command-line switches, and Chapter 4 discusses each one in detail.

Not enough memory
Turbo Debugger ran out of working memory while loading.

Old configuration file
You have attempted to start Turbo Debugger with a configuration file for a previous version. You must create new configuration files for this version of Turbo Debugger.

Remote link timeout
The connection to the remote system has been disrupted. Try rebooting both systems and starting again. If the problem persists, refer to Appendix E, where debugging on a remote system is discussed.

Unsupported video adapter
Turbo Debugger can't determine what display adapter you are using; MDA, CGA, EGA, VGA, MCGA, Hercules, Compaq composite, AT&T, and close compatibles are supported.

Wrong version of TDREMOTE
You have an incompatible version of TDREMOTE running on the remote system. You must use the same release of Turbo Debugger and TDREMOTE together.

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 **O**pen Log File command after having already issued the same command without an intervening **C**lose Log File command. If you want to log to a different file, first close the current log by issuing the **C**lose Log File command.

Ambiguous symbol name
You have entered a symbol name in an expression that does not uniquely identify a method in a C++ or object 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.

Assignment out of range
When doing a Pascal assignment, you have attempted to assign a value to a variable that is beyond the range of legal values for the variable.

Bad configuration file name
You have specified a nonexistent file name with the **–c** command-line option.

Cannot be changed
You tried to change a symbol that can't be changed. The only symbols that can be changed directly are scalars (**int**, **long**, and so forth in C; Byte, Integer, Longint, and Strings in Pascal) and pointers and strings in Pascal. If you want to change a structure or array, you must change individual elements one at a time.

Can't execute DOS command processor
Either there was not enough memory to execute the DOS command processor, or the command processor could not be found. Make sure that the COMSPEC environment variable correctly specifies where to find the DOS command processor.

Can't go resident until user program terminates
You have attempted to make Turbo Debugger resident before the program you are debugging has gone resident itself. Turbo Debugger can go resident only when there is no program loaded or when the loaded program has run and terminated.

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:ax
```

should have been

```
mov es:[bx],ax
```

Can't set a breakpoint at this address
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 have attempted to set a hardware condition on a breakpoint that is not 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.

Can't swap user program to disk
You issued a command that required the program being debugged to be written to disk, but there is no room on your current disk to write it. You will have to make some space on your disk before issuing any commands that require the program to be swapped. The File | DOS Shell and Edit commands in text panes both require the program to be swapped.

Can't use same register twice
You attempted to assemble an instruction that used a base or index register twice in the same memory operand. You can only use a register once in any operand. For example,

```
mov ax,[bx+bx]
```

should have been

```
mov ax,[bx+si]
```

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.

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.

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.

Edit program not specified
You tried to use the Edit local menu command from a Module or Disk File window, but you did not specify an editor startup command by using the installation program.

Error loading program
DOS was not able to load the program you specified. This could mean the file you specified is not a valid .EXE file, or that the .EXE file has been corrupted.

Error opening file ___
Turbo Debugger couldn't open the file that you want to look at in the File window.

Error opening log file___
The file name you supplied for the **O**pen 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 recording keystroke macros
An error occurred while writing the recorded macro keystrokes to the configuration file. The macro was probably not recorded to disk.

Error saving configuration
Turbo Debugger could not write your configuration to disk. Make sure that there is some free space on your disk.

Error swapping in user program, press key to reload
After swapping your program to disk to execute another program that you specified, Turbo Debugger is unable to reload your program. This most likely means that you accidentally deleted the disk file that your program was swapped to (SWAP.$$$). The only thing that the debugger can do is to reload your program exactly as if you had issued the **File | Open** menu command.

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
Turbo Debugger could not write your changes back to the file. The file may be marked as read-only, or a hard error may have occurred while writing to disk.

Expression accesses more than one scope
In conjunction with a breakpoint, you entered an expression that contains references to variables from too many scopes. In Pascal, you can reference local variables and parameters, globals, and locals from an outer subprogram (if the breakpoint is in a nested procedure or function). In C, you can reference function autos, module statics, and program globals, but not autos from more than one function.

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 64 operators and operands in an expression. Examples of operands are constants and variable names. Examples of operators are plus (+), assignment (= or :=), structure member selection (->), 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 Turbo Debugger 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.

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 three circumstances:

- You are attempting to call a function that is in a Pascal overlay.
- You are attempting to call a function while your current program location is in a Pascal 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 Turbo Debugger 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 far address
When entering an instruction to assemble, you supplied a badly formed far address for the target of a **JMP** or **CALL** instruction. A far address consists of a pair of hex numbers separated by a colon. For example,

```
JMP 1234:XYZ
```

should have been

```
JMP 1234:1000
```

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 parameters
You have attempted to call a function in an expression, but you have not supplied the proper parameters to the function 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 operand separator
You entered an instruction to assemble but didn't separate the operands with a comma. If an instruction has more than one operand, you must always use a comma between the operands. For example,

```
ADD ax 12
```

should have been

```
ADD ax,12
```

Invalid operand(s)
The instruction you are trying to assemble has one or more operands that are not 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 have 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 a expression that contained an incorrectly formed typecast. A correct C cast starts with a left parenthesis, contains a possibly complex data type declaration (excluding the variable name), and ends with a right parenthesis. For example,

```
(x *)p
```

should have been

```
(struct x *)p
```

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 (C and assembler only)
The C 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 one of the C structure member selectors (. or ->) or the Pascal record field qualifier (.). This symbol, however, was not preceded by a record or structure name, nor was it preceded by a pointer to a record or structure.

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, nor does the program you're debugging 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 Turbo Debugger could not find a relevant help screen. Please report this to Borland technical support.

No modules with line number information
You have used the View | Module command, but Turbo Debugger 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 325 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 **R**un menu can be performed without having a program loaded. Use the **F**ile | **O**pen command to load a program before issuing these commands.

No source file for module ___
No source file can be found for the module you want to view. If the source file is not in the current directory, you can use the **O**ptions | **P**ath for Source command to specify which directory your source file(s) are in.

No type information for this symbol
You have 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 have entered an expression that contains a function call, but the name preceding the left parenthesis introducing the function call is not a function name. Any time a parenthesis immediately follows a name, the expression parser presumes that you intend it to be a function call.

Not a memory referencing expression
, memory areas and
You have entered an expression that does not refer to a memory location. There are many cases where the expression must reference a memory location, not just return a value. For example, the **D**ata | **I**nspect command requires that the data item you inspect be a memory area, not just an expression with a result. For example,

```
3 * 4 < (9 - 1)
```

does not reference memory, but

```
myarray[4]
```

does reference a memory location.

Not an Object Pascal or C++ program

Your program is not an object Pascal or C++ program, so it does not contain any objects; therefore, command you selected cannot be performed.

Not a record, structure, or union member
You entered an expression that used one of the C structure member selectors (. or ->) or the Pascal record field qualifier (.). This symbol, however, was not preceded by a record or structure name, nor was it preceded by a pointer to a record or structure.

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. If your system has EMS memory, make sure that Turbo Debugger is set to use it for the symbol table. You can use TDINST to set it.

If you don't have EMS or your program doesn't load even with EMS, you can hook two systems together and run Turbo Debugger on one system and the program you're debugging on the other. See Appendix E for more information on how to do this. Or consider using TD286 protected-mode or TD386 virtual debugging. See Chapters 15 and 16 for more information.

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 Turbo Debugger uses when showing you your source code and program variables. If you have any resident utilities consuming memory, you may want to remove them and then restart Turbo Debugger. 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. If you're using TD386, try the **-f** option to force TD to emulate expanded memory. See Chapter15 for details.

When this message is issued, your program itself has not even been loaded. This means you must free enough memory for the symbol table and your program.

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. Some things that must reference memory include the assignment operators (**=**, **+=**, and so on) and the increment and decrement (**++** and **- -**) operators.

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
```

Overlay not loaded
You've tried to set a pane in the CPU window to a location in your program that is not presently loaded into memory. You can use a Module window to examine source code that has not yet been loaded into memory, but you can't look at the underlying instructions since they haven't yet been loaded into memory.

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 non-existent 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 symbol table
The program you want to debug has been successfully loaded, but it does not contain any debug symbol information. You'll still be able to step through the program using a CPU window and examining raw data, but you will not be able to refer to any code or data by name.

To create a symbol table in Turbo Pascal (5.0 or later), turn on Debug | Standalone Debugging (or use the **/v** command-line option with TPC.EXE). If you're using Turbo C or Turbo C++, you must compile with **/v** and link your program with TLINK, using the **/v** option, in order to get debug symbol information. If you're using Turbo Assembler, assemble with **/zi** and link with **/v**.

Program linked with wrong linker version
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 Turbo Pascal.

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.

Repeat count not allowed
You have entered a format control string that has a repeat count, but the expression that you are applying it to can't have a repeat count.

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
Turbo Debugger 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 **O**ptions | **P**ath for Source command
- in the current directory
- in the directory where Turbo Debugger found the program you're debugging

You should add the directory that contains the source file to the directory search list by using the **O**ptions | **P**ath for Source command.

Symbol not found
You entered an expression that contains an invalid variable name. You may have mistyped the variable name, or it may 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.

Type EXIT to return to Turbo Debugger
You have issued the **File | DOS** Shell command. This message informs you that when you are done running DOS commands, you must type `EXIT` to return to your debugging session.

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 (‘) in C.

Unknown record or structure name
You have entered an expression that contains a typecast with an unknown record, structure, union, or enum name. (Note that C and 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 (" in C, ' in Pascal) If you want to enter a string that contains quote characters in Pascal, they must contain additional quote characters ('). To enter a C string with quote characters, you must precede the quote with a backslash (\) character.

Value must be between 1 and 32
You have entered an invalid value for the tab width. Tab columns must be at least 1 column wide, but no more than 32 columns.

Value out of range
You have entered a value for a Pascal variable that is outside the range of allowed values.

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.

Video mode switched while flipping pages
Your program has changed the video display mode when Turbo Debugger is in page flipping mode. This means that the contents of your program's screen may have been lost. You can avoid this by using the **–ds** command-line option to set video swapping mode.

Information messages

Turbo Debugger generates some information messages that appear before the normal windowed display starts up. Here's a description of them.

TDREMOTE online
Turbo Debugger has succeeded in establishing communications with the TDREMOTE remote debug driver program on the remote system. If you specified a program name to load on the DOS command line, that file will now be loaded into the remote system.

Waiting for handshake from TDREMOTE (Ctrl-Break to quit)
You have told Turbo Debugger to debug your program on the remote system connected via the serial port (**-r**, **–rs**, and **–rp** command-line options). Turbo Debugger is now waiting for the remote system to inform it that it is running.

You can interrupt Turbo Debugger and return to the DOS prompt by pressing *Ctrl-Break*.

A P P E N D I X

G

Using Turbo Debugger with different languages

In this appendix, we have gathered together some tips on how to most effectively use Turbo Debugger with different languages.

Turbo C tips

Compiler code optimizing

If you have used the **–O** command-line option with TCC or the **O**ptions | **C**ompiler | **O**ptimization command with the Turbo C integrated environment to specify optimized code generation, you may have difficulty stepping through certain source code areas. In particular, if you have multiple or nested **if..else** statements, it may be difficult to stop as each **else** clause is encountered. A **for** loop is also rearranged in a manner which makes tracing through it a little odd in some situations.

To get around these (infrequent) problems, you can either switch to assembler-level debugging by opening a CPU window, or you can disable optimizing in the compiler while you are debugging.

Accessing pointer data

Many times in C, you use pointers to refer to arrays of data items. Normally, Turbo Debugger shows you the single pointed-to item when you inspect a pointer variable. To access a pointer as an array, you can first inspect the data item with one of the usual techniques, such as placing the cursor over the variable in a Module window and pressing *Ctrl-I*, and then set a range of items to look at by using the **R**ange command on the Inspector window local menu. For example, if your program contained

```
char *p, buf[80];
for (p = buf; p < buf + sizeof(buf); p++) {
   ...
}
```

you can examine *p* as an array of characters by choosing the **R**ange command in the Inspector window's local menu, and entering a starting index of 0 and a count of 80.

Stepping through complex expressions

If you have a complex expression, such as

```
if (isvalid(x) && !useless(x)) {
   ...
}
```

you may want to see the result of each subexpression that makes up the conditional expression. If there are function calls in the expression, press *F7* to trace into a function, put the cursor on the closing } at the end of the function, and press *F4* to run to that point. Then, choose the **D**ata | **F**unction Return command to look at the value about to be returned. If there are other function calls in the conditional expression, you can then press *F7* to stop on the first line of the next function in the conditional expression. You can then repeat this procedure to examine its return value.

If you have a complex expression that does not contain function calls, for example,

```
if (x <= 5 && y[z] > 8) {
   ...
}
```

and you want to see the result of evaluating each subexpression, you will have to open a CPU window, do assembler-level stepping, and watch the subexpression results being put in CPU registers.

Turbo Assembler tips

Looking at raw hex data

You can use the **D**ata | Add **W**atch and **D**ata | **E**valuate/Modify commands with a format modifier to look at raw data dumps. For example,

```
[ES:DI],20m
```

specifies that you want to look at a raw hex memory dump of the 20 bytes pointed to by the ES:DI register pair.

Source-level debugging

You can step through your assembler code using a Module window just as with any of the high-level languages. If you want to see the register values, you can put a Registers window to the right of the Module window.

Sometimes, you may want to use a CPU window and see your source code as well. To do this, open a CPU window and choose the Code pane's **M**ixed command until it reads Both. That way you can see both your source code and machine code bytes. Remember to zoom the CPU window (by pressing *F5*) if you want to see the machine code bytes.

Examining and changing registers

The obvious way to change registers is to highlight a register in either a CPU window or Registers window. A quick way to change a register is to use the **D**ata | **E**valuate/Modify command. You can enter an assignment expression that directly modifies a register's contents. For example,

```
SI = 99
```

loads the SI register with 99.

Likewise, you can examine registers using the same technique. For example,

```
Alt-D E AX
```

shows you the value of the AX register.

Turbo Pascal tips

Stepping through initialization code

When you first load your program into Turbo Debugger, the right-pointing filled arrow points to the **begin** keyword of the main program. The **begin** actually corresponds to a series of calls to the initialization sections of all the units that your program uses (assuming they have initialization code). All programs begin with a call to the initialization code of the *System* unit.

At this point, if you press *F7* (the hot key for the **Run | Trace** Into command), you'll trace into the the first unit that has initialization code with debug information enabled. If you use *F7* to step past the **end** of the first unit's initialization code, you'll trace into the next unit; eventually you'll return to the main program, ready to execute the first statement.

If, on the other hand, you press *F8* (the hot key for the **Run | Step** Over command) at the beginning of the program, you will skip over all initialization code and begin stepping through the body of the main program.

Stepping through exit procedures

When your program terminates, control is passed down a chain of exit procedures (refer to the chapter titled "Inside Turbo Pascal" in the *Turbo Pascal Object-Oriented Programming Guide*). When you step past the **end** of the main program, Turbo Debugger does not trace into the exit procedures. In order to step through this chain, place a breakpoint in each exit of the procedures you want to debug.

Constants

Constant identifiers are recognized only for scalar and typed constants; for example,

```
program Test;
const
  A = 5;
  B = Pi;
  Message = 'Testing';
  Caps = ['A'..'Z'];
  Digits : string[10] = '0123456789';

begin
  Writeln(A);
  Writeln(B);
  Writeln(Message);
  Writeln('A' in Caps);
  Writeln(Digits);
end.
```

In this program, you can inspect *A* (a scalar constant), *Digits* (a typed constant), *B* (a floating-point constant), or *Message* (a string constant), but not *Caps* (a set constant).

String and set temporaries on the stack

If you're using the CPU window, be advised that Turbo Pascal automatically allocates string and set temporaries on the stack in the following way:

The plus (+) operator, when used with strings, and all string functions will reserve stack space for results of these operations. This stack space is reserved in the caller's stack frame. Likewise, the +, –, and * set operators will also reserve stack space for intermediate results.

Clever typecasting

The *Dos* unit defines the internal data format for all the predefined file types. You can use these declarations to examine the data of any file variable. Try entering this program:

```
program Typecast;
uses Dos;
var
  TextFile : Text;
  IntFile : file of Integer;
begin
  Assign(TextFile, 'TEXT.DTA');
  Rewrite(TextFile);
  Assign(IntFile, 'INT.DTA');
```

```
    Rewrite(IntFile);
    Close(TextFile);
    Close(IntFile);
  end.
```

Now add these four watch expressions:

```
  IntFile
  TextFile
  FileRec(IntFile),r
  TextRec(TextFile),r
```

The first two will display the file status (CLOSED, OPEN, INPUT, OUTPUT) and disk file name, while the second two use typecasting to reveal internal field names and values for the file variables.

CPU window tips for Pascal

- Routines in the *System* unit are unnamed. When watching a call instruction in the CPU window, you will see a call to an absolute address instead of a symbolic name.
- A number of I/O routines (for example *Readln* and *Writeln*) often generate multiple assembler-language calls.
- Range-checking, stack-checking, and I/O-checking generate calls to library routines to perform their respective functions.
- A number of operators (Longint multiplication, string concatenation, and so on) are implemented via calls to library routines.
- The literal constants (string, set, and floating-point) of a procedure are placed in the code segment, just before the procedure's entry point.

G L O S S A R Y

The terms listed here are used frequently in this manual. Some of them are general terms about software and computers, and others are specific to the Turbo Debugger environment.

action	What happens when a breakpoint gets triggered. Actions can stop your program, log the value of an expression, or execute an expression.
active pane	The pane in the active window that is accepting user input. All cursor motion and local menu commands act upon this pane.
active window	The window on the display that the user is interacting with. Only one window can be the active window. It has its title in reverse video, and a double-line rather than a single-line border.
array	A data item composed of one or more items of the same data type.
ASCII	The native character set of the IBM PC and many other computers.
assembler	A form of machine instructions that humans can read, with opcode mnemonics. The Code pane of a CPU window lets you assemble instructions directly into memory.
autovariable	In the C language, this is a variable in a program that is local to an instance of a called function. These variables are stored on the stack, and their scope is that of the enclosing block (in C, source lines between a pair of { }).
block scope	The region of the program in which a specific data item is "visible." For example, some variables have *global* scope, meaning they are accessible anywhere in your program; other variables may be *local* to a module or procedure.
breakpoint	An address in the program you are debugging where some action is to be performed. See also *action*.
button	A dialog box item, represented by shadowed text, that executes a command or confirms settings you have made in the dialog box.

casting	Converting an expression from one data type to another. For example, converting from an integer to a floating-point number. In C, a cast consists of a data type enclosed in parentheses, like (*int*). In Pascal, a typecast consists of a type, followed by an expression surrounded by parentheses, like *word(5)*. (Also called typecasting and type conversion.)
C expression	An expression using the C language syntax. Turbo Debugger lets you evaluate any C expression, including those that assign values to memory locations.
check box	A dialog box item that toggles a setting between *On* and *Off*. When the option is set to *On*, an X appears between the square brackets of the check box: `[X]`.
CPU	The central processing unit; refers to the 80x86 processor in your system. The CPU has a number of flags and registers. The CPU window shows the current CPU state.
CPU flag	One of the control bits in the CPU that either affects subsequent instructions or is set to reflect the results of an operation.
CPU register	A fast storage location inside the CPU chip. The register names are AX, BX, CX, DX, SI, DI, BP, SP, CS, DE, ES, SS.
configuration file	A file in either the current directory or in the path that sets Turbo Debugger default parameters.
CS:IP	The current program location, as specified by the code segment (CS) CPU register, and the instruction pointer (IP) register.
default	A value automatically supplied when none is specified by the user.
dialog box	An onscreen box in which you can view and adjust settings and input information.
disassembler	A program that converts machine code into assembler code that you can read. The Code pane in a CPU window automatically disassembles instructions in one of its panes.
EMS	Expanded memory specification. Turbo Debugger can put your program's symbol table in EMS to conserve main memory.
expression	A combination of operators and operands conforming to the syntax of one of the languages supported by Turbo Debugger: C, Pascal, and assembler.
global breakpoint	A breakpoint that can occur on every instruction or source line.

history list	A list of previous user input lines maintained for each input box. This lets you select a previous entry instead of having to type it in.
inspector	A window used to examine or change the values in a data element, array, or structure.
local menu	The menu of commands that apply only to a particular window or pane. Press *Alt-F10* to pop up the local menu for the current pane.
menu bar	The bar at the top of the screen from which *pull-down menus* come. The commands on these menus are always available regardless of what you're doing in Turbo Debugger. Press the *Alt* key in combination with the highlighted letter of a menu bar item to access these menus.
operand	The data item that an operator acts on; for example, in *3 * 4*, both 3 and 4 are operands.
operator	An action that is performed on one or more operands, such as addition (+) or multiplication (*).
pane	A section of a window that contains logically related information. Panes can be scrolled independently of each other. When the size of a window is changed, its panes are adjusted to make the best use of the new window size. Each pane has a local menu of commands. See also *active pane*.
PATH	The DOS environment variable that indicates where to search for executable programs. Turbo Debugger searches the path for a configuration file.
pop-up menu	A menu that appears in midscreen, instead of pulling down from the menu bar.
postfix	An operator that comes after its operand, like *x++* in C.
prefix	An operator that comes before its operand, like *--x* in C.
pull-down menu	A menu of commands that pulls down from the menu bar.
radio buttons	A set of three or more options, one and only one of which must be active at any given time. If a radio button is on, a bullet appears between parentheses: `(•)`.
record	See *structure*.
reverse execution	The process of stepping backward through your program one instruction at a time, undoing the effects of program execution as you go.

scalar	A basic data type consisting of ordered components such as Byte, Integer, Char, and Boolean in Pascal or char, int, and float in C. Scalars can be the individual elements of larger data items, such as arrays or structures.
scope	See *block scope*.
set	An unordered group of elements, all of the same scalar type.
stack	The region of memory that stores procedure and function return addresses, parameters, and other data related to an instance of a called procedure or function.
side effect	An expression that alters the value of a variable or memory location; for example, an assignment statement or one that calls a function in your program that modifies some data.
step	To execute the program being debugged one instruction or source line at a time, while treating procedure or function calls as a single instruction. This lets you skip over calls to routines that you don't want to examine one line at a time.
structure	A data item composed of one or more elements of possibly dissimilar types.
symbol	A name of any variable, constant, procedure, or function.
trace	To execute a program one instruction or source line at a time.
tracepoint	A global breakpoint that watches for a variable or memory area to change.
triggered	A breakpoint is triggered when all the things controlling it become true: Your program must have reached the specified address, the pass count must have been reached, and the condition must have been satisfied.
type	Data items in your program have different types indicating their purpose. For example, your program can contain pointers, floating-point numbers, arrays, and so on.
watchpoint	A global breakpoint that watches for an expression to become true.
wildcards	The characters * and ?, used in file matching expressions. ? matches any single character * matches zero or more characters For example, `abc*.1` matches `abc99.1` and `abcdef.1` but not `xyz99.1`.

window A rectangular area of the screen containing information that can be viewed independently of the contents of other windows. In Turbo Debugger, windows can partially or completely obscure one another. See also *active window*.

I N D E X

2.0

USER'S GUIDE

TURBO DEBUGGER®

B O R L A N D

1800 GREEN HILLS ROAD, P.O. BOX 660001, SCOTTS VALLEY, CA 95067-0001, (408) 438-5300 ■ PART # 15MN-ASD01-20 ■ BOR 1489
UNIT 8 PAVILIONS, RUSCOMBE BUSINESS PARK, TWYFORD, BERKSHIRE RG10 9NN, ENGLAND
43 AVENUE DE L'EUROPE—BP 6, 78141 VELIZY VILLACOUBLAY CEDEX FRANCE